PROCEEDINGS

Workshop on UNIX and Supercomputers

Pittsburgh, PA

September 26-27, 1988

USENIX PROCEEDINGS 1988 Workshop on UNIX and Supercomputers

The USENIX Association

The USENIX Association is a non-profit association of individuals and institutions interested in fostering innovation and sharing ideas, software, and experience where UNIX and UNIX-like systems and the C programming language are concerned. USENIX sponsors technical conferences and workshops and an annual vendor exhibition; publishes *;login:* (a bi-monthly newsletter) and *Computing Systems* (a technical quarterly); distributes 2.10BSD and the 4.3BSD manuals; and serves as coordinator of a software exchange for appropriately licensed members.

The individual and institutional members of USENIX are interested in problem-solving with a practical bias, with research that works, and with timely responsiveness within a community made up of executives and managers, programmers, and academics.

Future Events

EUUG Autumn Conference
Estoril, Portugal, Oct. 3-7

C++ Miniconference
Denver, CO, Oct. 17-21

The Program Chair is Andrew Koenig of AT&T.

Large Installation Systems Administration II
Monterey, CA, Nov. 17-18

The Program Chair is Alix Vasilatos of MIT's Project Athena.

Japan UNIX Society
Osaka, Nov. 11-15, Conference & Exhibition
Toyko, Dec. 7-8, UNIX Fair '88

For both events, contact: Ms. Hiroko Tsunoda, Japan UNIX Society, 2-12-505 Hayabusa-cho, Chiyoda-ku, Tokyo 102, +81-3-234-5058

USENIX 1989 Winter Technical Conference
San Diego, Jan. 30-Feb. 3, 1989

EUUG Spring Conference
Brussels, Apr. 10-14, 1989

Long-term USENIX & EUUG Schedule

Jun 12-16	'89 Hyatt Regency, Baltimore
Sep 18-22	'89 Vienna, Austria
Jan 22-26	'90 Omni Shoreham, Washington, DC
Apr 23-27	'90 Munich, W. Germany
Jun 11-15	'90 Marriott Hotel, Anaheim
Jan 21-25	'91 Dallas
Jun 10-14	'91 Opryland, Nashville
Jan 20-24	'92 Hilton Square, San Francisco
Jun 8-12	'92 Marriott, San Antonio

© 1988 USENIX Association, All Rights Reserved.
This volume is published as a collective work.
Rights to individual papers remain
with the author or the author's employer.

UNIX is a registered trademark of AT&T.

Program and Table of Contents

Workshop on UNIX and Supercomputers

September 26-27, 1988

Sunday, September 25, 1988 — Sky Room, 17th floor

Registration and No-host Reception 6:00 - 9:00

Monday, September 26, 1988 — Urban Room

Opening Remarks and Announcements 9:00 - 9:15

Processes and Process Management 9:15 - 10:15

Break 10:15 - 10:45

Experiences 10:45 - 12:00

Lunch and Birds of a Feather Sessions 12:00 - 1:45

Performance 1:45 - 3:00

Tuesday, September 27, 1988 — Urban Room

Conference Co-Chairs:

Lori Grob
NYU Ultracomputer
Research Laboratory
715 Broadway, 10th Floor
New York, NY 10003

(212) 998-3339
grob@lori.ultra.nyu.edu

Melinda Shore
Frederick Cancer
Research Facility
P.O. Box B, Building 430
Frederick, MD 21701

(301) 698-5660
shore@ncifcrf.gov

USENIX Conference Coordinator:
Judith H. DesHarnais

USENIX Conference Liaison:
Deborah K. Scherrer

Proceedings Production:
Tom Strong
Ellie Young

NOTES

Process Management for Highly Parallel UNIX Systems

Jan Edler, Jim Lipkis, and Edith Schonberg

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
New York University
715 Broadway, 10th floor
New York, NY 10003

ABSTRACT

Despite early development exclusively on uniprocessors, a growing number of UNIX systems are now available for shared memory (MIMD) multiprocessors. While much of this trend has been driven by the general success of the UNIX interface as an emerging industry standard, experience has shown that the basic UNIX design is amenable to such environments. Relatively simple extensions such as shared memory and synchronization mechanisms suffice for many parallel programs.

While simple needs can be satisfied in a simple fashion, the desire to support more sophisticated applications has created pressure for ever more complex extensions. Is there a better way to meet such needs? Although some argue that it is time to abandon the UNIX model completely, we believe that viable alternatives exist within the traditional framework.

In this paper we propose several modifications to the process management facilities of the UNIX kernel. Some of them are primarily of interest for parallel processing, such as a generalized *fork* system call that can efficiently create many processes at once, while others are equally attractive in other contexts, such as mechanisms for improved I/O and IPC performance. While the primary goals are improved performance and reliability, a strong aesthetic judgement is applied to create a total design that is cohesively integrated.

While the concepts presented here are applicable to any UNIX environment, they have been conceived in the context of very large scale parallel computing, with hundreds or thousands of processors. An initial implementation of these extensions is currently underway for the NYU Ultracomputer prototype and the IBM RP3.

This work was supported in part by I.B.M. under joint study agreement N00039-84-R-0605(Q), and in part by the Applied Mathematical Sciences Program of the U.S. Department of Energy under contract DE-FG02-88ER25052.

1. Introduction

A number of versions of the UNIX[1] operating system have been constructed for various shared memory (MIMD) parallel processors, with kernel interface semantics spanning a range from those with relatively minor extensions to those representing a radical departure from tradition. It is possible to support many parallel applications with only a single extension to the basic set of system calls, to provide for shared memory. In fact, a number of versions of UNIX for uniprocessors, such as System V [AT&T86], have had this feature for years. Unfortunately, the range of parallel applications that can be efficiently supported on such a minimally extended system is rather narrow, leading to an assortment of further extensions, such as:

- Various non-busy-waiting synchronization mechanisms, e.g. locks, semaphores, events, etc.
- Different kinds of processes with more efficient semantics than standard processes.
- New representations of process state for executing, debugging, core-dumping, and/or checkpointing parallel programs.
- Enhancements to the scheduler to improve performance of parallel jobs.
- Enhancements to I/O and IPC performance.

With demand for new extensions like these increasing the complexity of the system, the idea of adopting a new system model and starting with a fresh design (free from the burden of existing UNIX semantics) seems like an attractive alternative. Unfortunately, a completely new design loses many of the *benefits* of UNIX as well, including compatibility and familiarity. Adopting UNIX compatibility as a constraint on a fresh design simply leads to *even more* complexity, from the inevitable proliferation of similar-but-different mechanisms and numerous "special cases" required to hold the whole thing together.

In this paper we discuss a set of modifications, primarily in the area of process management, that we believe form a natural generalization and build upon the strength of existing UNIX facilities, in order to provide efficient support for sophisticated parallel applications in a system with manageable complexity. Although they differ substantially from the old facilities in some respects, compatibility and implementability have been carefully considered. Despite the fact that these proposals have been conceived and developed for highly parallel machines (with potentially thousands of processors), such as the NYU Ultracomputer [Gottli87] and the IBM RP3 [PBG85], we have tried not to lose sight of smaller machines and uniprocessors as well.

The proposals in this paper are generally made in the context of the Symunix II operating system, currently being designed and implemented at NYU. We believe that they form a cohesive whole when taken together with the memory management interface described in [ELS88].

[1] UNIX is a registered trademark of AT&T.

We begin the next section by considering mechanisms to control parallelism, that is, how to create and manage multiple instruction streams, and discuss a spectrum of implementation models which the new design can support. This leads into the following section where control of processor scheduling is addressed. In section 4, asynchronous system calls, important for the support of highly-efficient user-mode multitasking as well as high performance I/O, are introduced. We then go on in section 5 to address synchronization and IPC issues, and extend the signal mechanism in section 6. We conclude with a summary and closing arguments in section 7.

2. Parallelism

The first question that must be asked when considering MIMD processing in a UNIX environment is what form do the multiple instruction streams take, how are they created, destroyed, and managed?

2.1. Processes

Given some form of shared memory, an acceptable environment for many parallel applications can be constructed with the established **fork**, **wait**, and **exit** system calls, where each instruction stream is represented by a UNIX process. While this approach is relatively easy to implement, there are several reasons why it is inadequate for any but the simplest of environments:

- Since processes must be created and waited-for one at a time, **fork** and **wait** represent a potential serial bottleneck. Whether or not this bottleneck is severe depends on the number of processes to be created and the proportion of the total run time of an application that is consumed by **fork** and **wait**. As an improvement over serial process initiation, a logarithmic approach may be used in which each new process helps create more processes, but this implies more complexity and overhead, and has another problem discussed in the next bullet:
- Processes created by **fork** form a hierarchy, which can be awkward within a single parallel application where no such hierarchical relationship logically exists. The number of processes assigned to an application may grow and shrink over time, not according to block structured nesting, but according to the amount of parallelism in an application during various phases of execution and the availability of processors on the machine. This would not be a problem except for the way process termination is reported to the parent (or **init**, if the parent has already terminated) via the **wait** system call and an optional signal.
- There is insufficient control over the kinds of state changes which cause **wait** to return information. For example, it is not possible to ignore normal terminations and wait only for abnormal ones.

In addition, it is obvious that **fork** involves far too much overhead (even with the best of implementations) to be used to create the multiple instruction streams of parallel programs, in particular those with fine-grained parallelism. Because of this, many designers have questioned the very nature of the UNIX "process" abstraction, and its suitability as a vehicle for parallel applications. There are a

variety of proposals, including the threads of MACH [Tevani87], Topaz [Thacke88], or Stellix [Teixei88], UNICOS's tfork operation [Reinha85], and others [Kepecs85], but here we will simply refer to any of them as *lightweight processes*, because a primary goal is to reduce the "weight" of processes by removing some attributes such as private address space, open files, signals, current directories, etc., normally necessary to support UNIX semantics. All of this "unnecessary weight" contributes to the overhead of process creation, context-switching, etc.

There are three common misconceptions about lightweight processes that we wish to address:

Simplicity

It is often claimed that simplicity is gained by shedding a multitude of unnecessary features associated with the UNIX process. This would probably be true except for the fact that most of these features are *desirable*, making it necessary to re-implement them in some other way, either in the kernel or in some combination of libraries and server processes. In either case, it is quite possible that the overall result (once the discarded UNIX semantics are re-implemented) will add complexity to the implementation rather than subtract from it. The size and complexity of the programmer's interface will also increase, with the addition of lightweight process semantics. For example, elimination of the private address space is incompatible with established UNIX semantics, so most proposals simply add an optional shared address space concept to the system: an additional kernel feature certainly doesn't *reduce* overall complexity.

Efficiency

Lightweight processes can be created and destroyed more quickly than heavier traditional processes. However, no form of kernel-supported process (light or heavy) will be efficient enough to be created and destroyed on demand in fine-grained parallel applications. And in sufficiently coarse-grained applications the creation/destruction expense will not be an issue. There are no data yet on how many programs fall "in between" these two extremes, but in almost all cases the most efficient way to implement parallel applications is to create processes (light or heavy) in advance of need and schedule application-defined instruction streams to those processes without kernel intervention. Context-switching between lightweight processes that are part of the same application can be done more efficiently than between heavyweight processes, but it can be done even faster without kernel intervention. Implementation of kernel-supported lightweight processes would therefore represent an unnecessary duplication of function.

Demand Paging

When virtual memory techniques such as demand paging are a requirement, it may seem that lightweight processes implemented outside the kernel are not viable: if each instruction stream is a separate process, the kernel can schedule another one when a page fault occurs, but if streams are scheduled to processes in user-mode (as advocated above) that isn't possible. The same situation would arise if page faults were handled by hardware suspension of the faulting processor, without an interrupt to trigger a reschedule. It is possible to use the same solution outside the kernel: a signal can be delivered to

notify a process of certain paging events (e.g. page fault, page input complete, page eviction). Provided certain pages are "locked" in memory (to prevent recursive page faults in signal handling and instruction stream management), then the process can switch to another stream. Assuming a moderate page fault rate, the performance gain from scheduling outside the kernel will outweigh any additional overhead from signal handling.

For these reasons we believe lightweight processes offer insufficient advantages to justify their addition to the system as a basic kernel-supported facility. Instead, we extend the traditional process model, and rely somewhat more upon standard library and language runtime support software than most previous UNIX kernels, to provide superior performance outside the kernel.

2.2. The Process Control Bottleneck

Our first extension is conceptually simple: we modify **fork** to create more than one new process at a time. The enhanced **fork** is called **spawn**, and looks like this:

spawn (**int** n, **int** flags, **int** *pids)

The parameter *n* tells how many new processes to create, and *pids* points to a suitably large integer array, where the process ID of each child is stored. The return value is zero in the parent and a unique *spawn index* chosen from the set {1,...,*n*} in each child, such that, for each child *i*,

$pids[\text{spawn\_index}_i - 1] = \text{pid}_i$.

We define the term *family* to refer to a set of processes related by **spawn** but not **exec**; i.e. a single parallel program, with the various processes presumably cooperating. The original member of the family (the only one that has ever done an **exec**) is known as the *progenitor*. The process hierarchy awkwardness described in section 2.1 is addressed by providing a **spawn** option (a flag bit) that makes the progenitor the effective parent of all processes in the family, thus collapsing the hierarchy. If the option is set, attributes such as address space[2], open files, etc. are derived from the true parent, but SIGCHLD signals generated by the children are sent to the progenitor. The call **getppid** in the children will return the process ID of the progenitor.

The notion of progenitor ensures that child process status will not be lost if a parent process terminates — provided, of course, that the progenitor itself doesn't terminate. The addition of a new signal type, SIGPARENT, with default action "ignore", allows children to be notified in case of unexpected death of their parent (or progenitor). This is important for parallel applications that synchronize, since a process terminating unexpectedly (perhaps due to a program bug, such as division by zero) without notification to the rest of the family can deadlock the remaining processes.

What happens to orphaned children of parents and progenitors when SIGPARENT is ignored? In UNIX, when a parent terminates without waiting for all of its children to terminate, they are "inherited" by the **init** process. This is a slight

[2]Shared memory is provided by allowing shared memory segments to be inherited on spawn and also through file mapping, as described in [ELS88].

convenience to some UNIX kernels, but we are unaware of any version of **init** that actually depends upon this behavior. Therefore, for the sake of efficiency, we have chosen to simply not send SIGCHLD for orphaned processes.

Rather than generalizing the **wait** system call with new options to specify just which processes to wait for, we have chosen to eliminate **wait** in favor of an enhanced SIGCHLD signal. We defer a discussion of other signal-related issues to section 6, but for now the reader should assume that SIGCHLD signals will be sent to the parent for each "interesting" state change of a child (e.g. termination), that these signals are queued so that none are lost, and that SIGCHLD is accompanied by additional data passed as extra arguments to the parent's signal handler, providing essentially the same information as that obtained by **wait**.

Using the above facilities, it is not hard to build compatible versions of **fork**, **wait**, and a SIGCHLD handler that maintain a list of processes not yet waited-for. The only non-obvious aspects of the job are to properly handle the case where the parent calls **exec** before waiting for all its children, and to support old software that uses SIGCHLD with the old semantics, but it can be done.

What advantage does this scheme have, other than eliminating one system call? The parent can specify exactly which kinds of state changes are of interest by or'ing together certain flag values when calling **spawn**, and the type of each state change is included in the extra arguments passed to the signal handler. The following state changes can be selected:

Termination by signal
Termination by **exit**(0)
Termination by non-zero **exit**
Stopped by signal
Continued by signal

When a child changes to a state selected as interesting by its parent, SIGCHLD with appropriate arguments will be sent. The role of the zombie process is replaced by the queued signal mechanism. Most parallel programs will only be interested in abnormal terminations (termination by signal and non-zero **exit**).

2.3. Implementation Models

Given a fixed kernel interface, the implementor of parallel language environments is still faced with a large number of options, or tradeoffs, with significant impact on the functionality and efficiency of the resulting environment. At one extreme, the kernel interface described in this paper is available to the experienced C programmer, and can be used directly. While such an "environment" is extremely flexible, it is also very primitive. At the other extreme, languages with explicit parallel constructs require implementations with runtime support software to map the language-supported abstractions into the kernel-provided ones.

As has already been mentioned, using **fork** and **spawn** directly as the method of creating new instruction streams is too expensive for all but the most static types of parallel applications. It is more efficient for language-specific runtime support software (executing in user-mode) to schedule application-specific instruction streams to processes that have been created in advance; we refer to such processes as *prespawned*, and to the instruction streams as *tasks*. If tasks don't

necessarily run to completion, but may *block* during synchronization to allow other tasks to run on the same process, we say the runtime support system is *multitasking*.

A tasking system that schedules prespawned processes duplicates certain operating system functions, such as context switching, task creation/destruction, etc. However, depending on the task semantics being supported, the user-mode system can be orders of magnitude more efficient than an operating system. The efficiency/generality tradeoffs of tasking systems are illustrated by the following:

- If multitasking (blocking) is not needed, then the tasking system can use an extremely efficient run-until-completion scheduling paradigm. For many scientific applications written in Fortran extended with a parallel loop construct, such a simple scheduling paradigm is adequate.
- If tasks not only block, but must be *preempted*, a much more sophisticated multitasking system is necessary. The Ada[3] language, for example, requires a scheduler that performs preemption according to task priorities.
- If tasks never make requests for I/O or other system services, then the multitasking system can be simpler and more efficient.
- One might imagine supporting full UNIX process semantics in user-mode tasks, but this is difficult, expensive, and not generally useful. For example, signals with full UNIX semantics require process IDs, which would be hard to simulate in the fullest generality for tasks if they must interact with real (e.g. unrelated) processes. Operations which manipulate the address space in an incompatible way (e.g. exec) would be pointless, because context-switching in user-mode between tasks with incompatible memory images would be even more expensive than context-switching in the kernel.

It is important to note that, although the choice of semantics supported by a particular parallel language environment will have a profound effect upon the complexity and efficiency of the language runtime implementation, the nature of the underlying kernel interface will have much less effect. Techniques such as prespawning and user-mode multitasking remain important regardless of whether the kernel supports lightweight processes, or the way in which important system components (e.g. the file system, virtual memory system, network protocols, etc.) are implemented (e.g. inside or outside the kernel).

3. Scheduling

Scheduling has traditionally been transparent in UNIX, that is, with the exception of the **nice** system call (**setpriority** in 4.2BSD), there have been essentially no scheduling controls available to the programmer or system administrator. Scheduling is performed based on priorities that are adjusted dynamically according to resource usage and system load. In general, that is good; unnecessary controls on a system should be avoided. However, with the presence of parallel applications, new factors emerge that make some change increasingly necessary:

[3]Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

- *Frequent synchronization between processes.* From the point of view of a single parallel application running on a multiprocessor, the most efficient way to perform such synchronization is to run all synchronizing processes simultaneously, and allow them to *busy-wait* as necessary during the synchronization. Most parallel architectures directly support an efficient hardware mechanism to do this, avoiding the overhead associated with blocking a process and context switching to another one. Unfortunately, even on a multiprocessor there aren't always enough processors, so some form of process preemption is often still needed. However, busy-waiting in the presence of preemption can be very bad for performance. For example, a process might busy-wait a substantial amount of time for another process that has been preempted, creating a feedback effect by performing useless work that effectively raises the system load. This problem arises because the scheduler isn't aware that synchronization is going on [Zahorj88].

- *Uneven rates of execution.* Some parallel algorithms perform poorly when the instruction streams don't progress at approximately the same rate. While even progress is a property of any "fair" scheduler over a sufficiently long period of time, some algorithms have less tolerance and require fairness over shorter time periods. An example is an algorithm that uses *barrier synchronization* [Jordan78] frequently. In such a situation, the progress of the overall algorithm is limited by the progress of the last process to reach each barrier.

- *Complete program control over resource scheduling.* Beyond the problems we have just described, where lack of control over preemption can cause serious performance problems, there are applications that ask almost nothing of the operating system at all, except that it not interfere. What is often wanted by the user for these applications is a long-term commitment of resources (e.g. processors, memory), and, given that they are willing to pay the price for such commitment, they expect to get performance near the absolute maximum deliverable by the hardware, with no operating system overhead. Such control may not be appropriate on small machines or uniprocessors, but becomes increasingly important on large high-performance multiprocessors.

The common theme running through all these problems is preemptive scheduling of processes, and we now propose two new mechanisms to eliminate or control it: *scheduling groups* and *temporary non-preemption.*

3.1. Scheduling Groups

We introduce a new process aggregate, the *scheduling group*, which is identified by a *scheduling group ID.* Every process is a member of a scheduling group, although that group may consist of only the single process itself. In many cases, we expect scheduling groups to correspond to families, as defined in section 2. A scheduling group is subject to one of the following scheduling policies:

fully-preemptable
: This is the traditional scheduling policy of UNIX, and is the default.

group-preemptable
: The members may only be preempted or rescheduled at (approximately) the same time.

non-preemptable
The members are immune from normal preemption.

Our notion of group-preemption is similar to Ousterhout's coscheduling of task forces [Ouster82], but is concerned with preemption rather than blocking.

Since it is necessary for all members of a group-preemptable or non-preemptable group to execute simultaneously, it must be possible to meet their combined resource requirements. It is obvious that uniprocessors can't provide these new scheduling policies for groups containing more than one process, but multiprocessors also have a finite supply of processors and memory, and since spawning a new process into such a group or requesting more memory are operations that increase the resource requirements of the group, it is possible for them to fail. To avoid this problem, we must provide for long term allocation of processors and memory. Long term resource allocations, group-preemption, and non-preemption are restricted by system administrative policy and implemented by a long-term component of the scheduler. In addition to new system calls for manipulation of scheduling group IDs, new calls are needed to select the scheduling policy of a group, and request long-term allocation of processors and memory.

These new mechanisms directly support multiprocessor users whose primary objective is to "get rid" of operating system overhead, and solve the other problems mentioned above, provided that the users are willing to pay the cost which may include restrictions on resource utilization, increased average turnaround time, and in some cases a negative impact on interactive or real-time responsiveness.

The group-preemptable and non-preemptable scheduling policies have no effect on voluntary blockages (e.g. when a process issues a system call for which it is necessary to block). When a process is blocked, other members of the group will continue to be run, preempted, and rescheduled according to the current policy. Note also that the scheduling policy does not necessarily affect interrupt processing, so a process may still suffer the overhead of interrupts caused by some other process doing I/O; this is an unavoidable overhead of multiuser activity on many architectures.

3.2. Temporary Non-Preemption

Group- and non-preemptable scheduling policies are conservative mechanisms for users who need long-term control over preemption; as such they are expensive because of the long-term commitments required. There are other situations, in which synchronization occurs for short periods of time, that can be satisfactorily handled by a cheaper mechanism; we introduce the notion of *temporary non-preemption* for this purpose. This is a "hook" into the scheduler to allow processes to accommodate very short-term conditions. An example of typical usage is to prevent preemption while holding a short-duration busy-waiting lock. The process essentially says, "I don't mind being preempted, but please wait a bit until I tell you it's ok". The mechanism provides protection against malicious processes that never say "ok". Unlike the scheduling policies described above, it is useful on uniprocessors as well as multiprocessors. Two calls are provided, **tempnopreempt** and **tempokpreempt**. The first notifies the scheduler that temporary abeyance of preemption is desired, and the second signals a return to normal scheduling.

Much of the usefulness of this feature derives from the fact that the implementation is extremely efficient. The scheduler maintains for each process the address of a two word vector in the user address space; this address is set via a new system call normally issued by the language start-up code immediately after exec. The first word is for communication from the user to the scheduler, the second is for communication from the scheduler to the user. Initially, both words are set to zero. What **tempnopreempt** does is essentially

```
word1++;
```

while **tempokpreempt** is approximately

```
if (--word1 == 0 && (temp=word2) != 0) {
        word2 = 0;
        yield();
        return temp;
}
return 0;
```

Yield is a new system call that reschedules the processor. Because *word1* is incremented and decremented, rather than simply set, **tempnopreempt** and **tempokpreempt** can be safely nested. What the scheduler does is complementary; when it wants to preempt a process, code such as the following is executed on the same processor (i.e. the scheduler runs as an interrupt):

```
if (word1 == 0)
        ok to preempt;
else if (preemption not already pending for this process) {
        word2 = 1;
        note preemption pending;
}
else if (preemption pending for at least tlim time) {
        word2 = 2;
        ok to force preemption;
}
```

Where *tlim* is perhaps a few milliseconds, and the *preemption pending* state is maintained per-process and cleared on actual preemptions and on **yields**. The purpose of the **tempokpreempt** return value is to notify the user (after the fact) if preemption was requested or forced.

Overhead for this implementation is only a few instructions in the common case where no preemption would have occurred anyway, and only the overhead of the **yield** otherwise. The most abusive a user can get with this scheme is to lengthen a time-slice by *tlim*; it is easy to prevent any net gain from such a strategy by shortening the next slice by *tlim* to compensate.

4. Asynchronous System Calls

The basic UNIX I/O model of open/close/read/write/seek operations on byte streams is simple, powerful, and quite appropriate for the workload to which UNIX systems have traditionally been applied, but in higher performance environments some extensions are often necessary. One such extension is asynchronous I/O, which has existed in non-UNIX systems for decades and even within the UNIX kernel since its inception, but only fairly recently has it begun to become directly

available to the user, on systems such as the Cray [Cray86] and Convex [Convex87]. Of course it is possible to obtain asynchronous I/O on any UNIX system that has shared memory, simply by using another process to issue the synchronous I/O call. Our objections to such an approach are that it is less efficient since additional context switches are required, and it is more complex for the user software.

In large-scale parallel systems, user access to asynchronous I/O is desirable not only for high bandwidth and increased overlapping of processing and I/O activity, but also to facilitate the construction of efficient user-mode multitasking systems, as described in section 2. In such an environment, when a task does a synchronous read or write system call, the process executing the task can become blocked, thus "wasting" a process or processor if other tasks are ready to run; a better approach is to use asynchronous I/O, either directly, in the application, or indirectly, in runtime support software implementing the standard synchronous I/O model for tasks.

4.1. Meta System Calls

Rather than propose individual extensions to allow asynchrony in certain designated I/O operations, we propose a general strategy that can be applied to any system call for which asynchronous operation makes sense, e.g. not only **read** and **write**, but also **open**, **close**, **ioctl**, **mkdir**, **rmdir**, **rename**, **link**, **unlink**, **stat**, and so forth. The general idea is to define a control block for asynchronous system calls, identifying the specific call and giving the arguments. Additional fields are provided within the control block for returned values and status (e.g. pending, successfully completed, or completed with an error). Three new *meta system calls* are introduced, each taking a control block pointer as an argument:

syscall
: Issue a system call. The return value indicates whether or not the call has already completed, if not a signal (SIGSCALL) will be delivered when it is.

syswait
: Wait for completion of a system call. The return value distinguishes between control blocks that aren't pending, those that have just completed, and those that have been cancelled.

syscancel
: Cancel a pending system call, if possible.

The **syscall/syswait/syscancel** framework relieves the programmer from having to know exactly which calls are asynchronous and which aren't. It is even possible for a given call to sometimes complete synchronously and sometimes asynchronously; the **syscall** return value lets the user know.

In some cases, new calls or modified semantics must be introduced to make the most of asynchrony. For example, versions of **read** and **write** can be provided with explicit file position arguments, eliminating the need for separate **lseek** calls. This is important for asynchronous I/O since operations can be completed in any order.

Asynchronous system calls require queued signals with arguments. When SIGSCALL is delivered, the signal handler is called with an extra argument, a pointer to the completed control block. Instead of requiring all uses of asynchronous system calls within a program to use the same signal handler, we allow the user to specify a different handler in each control block.

4.2. Higher Level Interfaces

As a matter of convenience, we expect the development of a higher level interface to the asynchronous system calls along the lines of

aread (**int** fd, **void** *buf, **ulong** count, **ulong** pos, **void** (*func)());

where the function pointed to by the formal argument *func* is called on completion with a pointer to the actual control block, so that the various fields may be inspected as necessary. The implementation of such a higher level interface is merely a matter of allocating the control block, issuing the **syscall**, and after the completion signal is delivered, calling (**func*) and finally deallocating the control block.

In V7 and similar UNIX implementations, when a signal arrives during a "slow" system call (such as **wait**, **pause**, or some cases of **read** or **write** on "slow devices"), and a handler has been set for the signal, the system call is interrupted. If the handler returns, things are arranged so that the interrupted system call appears to return a special error, EINTR. Because it is usually inconvenient to deal with EINTR, systems compatible with Berkeley UNIX will automatically restart the system call in some cases. The introduction of asynchronous versions of all interruptible system calls allows us to avoid the problem altogether, by simply not interrupting the system call. If the handler really wants to interrupt a system call, it can do so by invoking **syscancel**. Implementation of the standard synchronous interface calls as a higher level interface above **syscall**, while not required, is fairly straightforward although there is some impact in machine-dependent signal-support code, such as the *trampoline* code and *longjmp*.

5. Synchronization and IPC

Shared memory multiprocessor architectures support a variety of underlying synchronization mechanisms. In addition to the common test-and-set and compare-and-swap primitives, these include tagged-memory schemes (e.g. full/empty bits) [Smith81] as well as generalized atomic read-modify-write operations such as fetch&add [Gottli87, PBG85]. Parallel programs, on the other hand, employ a wide variety of high-level synchronization functions. For maximum efficiency it is necessary to implement these directly in terms of the available hardware features. For example, simple mailboxes are well served by full/empty tags, while counting semaphores, barrier synchronization, and readers/writers locks (among others) are simply and efficiently implemented with fetch&add; of course, any of these architectural mechanisms can support simple mutual exclusion. These and other coordination functions can be implemented with minimal expense by taking advantage of the fact that in the common cases where no contention or need for delay exists, only a small number of instructions are required.

5.1. Process Management for Synchronization Functions

We believe that semaphores and other synchronization facilities should be provided in library and language runtime code. Kernel implementations (e.g., semaphores in System V) [AT&T86] will necessarily limit flexibility and discourage efficient exploitation of available hardware features. The role of the kernel should be limited to the management of process state, while communication and coordination among processes are implemented in user-mode. On multiprocessors, both busy-waiting and process-switching modes of synchronization are needed; the former to minimize the overhead of extra context switches, the latter to minimize idle processor time. A hybrid of busy-waiting and process-switching (also known as "two-phase blocking") [Ouster82] can combine the performance advantages of the two modes. Busy-waiting requires no support from the operating system, though the group scheduling policies of section 3.1 and the temporary non-preemption facilities discussed in Section 3.2 are useful for avoiding performance problems that can occur when process scheduling interacts with synchronization. Process-switching, on the other hand, requires system services for process suspension and resumption that are not subject to race conditions.

Several solutions have been proposed to this problem. The key to our mechanism is a per-process *pending unblock* flag maintained by the kernel as part of the process state. A new **block** system call atomically tests the *pending unblock* flag, and, if set, clears the flag and returns immediately; otherwise, it suspends the process. The **unblock** system call atomically tests whether the indicated process was suspended by a **block**, and, if so, makes it ready; otherwise it sets the *pending unblock* flag for the process. Reliable (i.e., race free) synchronization routines are possible because an **unblock** is effective even when it happens to occur before the target process has issued the corresponding **block**. For efficiency, **block** and **unblock** are avoided completely in the most common contention-free case. Furthermore, no other locking is needed when the primitive hardware-supported synchronization facility is suitable. This is important for cases like counting semaphores and readers/writers locks where it is possible to completely avoid serialization as long as blocking is not logically required.

MACH provides a different but equivalent mechanism for reliable suspension and resumption of threads, though it requires three system calls rather than two [Tevani87]. In UNIX, the same function can in fact be provided using the standard signal mechanism (**pause** and **kill** system calls). However, there are efficiency and operational drawbacks to this approach, including the requirement for a new, dedicated signal type.

5.2. Low Overhead IPC

Message passing remains a useful paradigm on shared memory architectures. Systems such as Accent [Fitzge86] have demonstrated the efficiency to be gained in certain situations by utilizing copy-on-write techniques when passing large messages. We would like to exploit memory sharing to provide very low overhead communication channels for small messages as well. Stream-oriented IPC applications can be implemented in the runtime library so as to execute almost entirely without kernel intervention once shared buffers are established.

Producer/consumer coordination, using the above synchronization facilities, ensures correct operation at very low cost.

A user-mode implementation of UNIX pipes seems particularly straightforward because communicating processes necessarily share a common parent. Memory regions that survive exec, as supported by the Symunix II memory management system [ELS88], can be allocated by the **pipe** "system call" for use as shared communication buffers. Library code (primarily in the **open**, **close**, **read**, and **write** routines) then implements the standard pipe semantics. The only remaining difficulty, requiring kernel support, is the SIGPIPE signal. With a general per-open-file flag distinguishing logical "readers" from "writers", a signal could be generated when the last "reader" maps out or closes the *image file* [ELS88] representing the shared buffer.

6. Signals

Since the main focus of our work has been on shared memory MIMD machines, we regard signals as playing a supporting rather than a central role in the system. That is, we are not primarily concerned with the use of signals for general IPC, but as a mechanism to support other facilities. Nevertheless, when modifications are required, we seek general solutions.

6.1. Signal Queuing

In order to adequately support enhanced child state-change notification (discussed in section 2) and asynchronous system calls (discussed in section 4), we need to queue pending signals that are to be caught. This means that all signals sent will be separately received with no omissions or duplications, and signals of a given type will be received in the order sent. Counting pending signals would be simpler, but doesn't allow additional data to accompany the signal (e.g. child status or pointer to control block). While it is not necessary that queuing be introduced for other signal types, it seems worthwhile to have similar behavior for all signals (and we doubt the existence of many applications depending on non-queued behavior). The only signals for which queueing makes no sense at all are signals that aren't catchable or maskable, such as SIGKILL and SIGSTOP. Also, certain signals sent by the kernel in response to synchronous events (e.g. SIGILL, SIGFPE, SIGSEGV), probably should not be maskable (or sendable by system call) and therefore might as well not be queued.

Because of signal queuing, a new resource type (a "signal delivery structure") must be introduced into the kernel, and it is possible (although unlikely) for this resource to be depleted, preventing a signal from being sent. A partial solution to this problem is to pre-reserve delivery structures as soon as the possibility of future need arises. For example, when a process is created, structures for SIGCHLD and tty-generated signals can be reserved. This approach can be used together with other techniques such as multiplicity counts (to compactly represent many consecutive identical signals) and possibly a limit on the length of the per-process pending signal queues.

6.2. The Word Size Barrier

Extending the number of signal types to more than the number of bits in an **int** causes problems with **sigblock**, **sigpause**, and **sigsetmask**; 4.3BSD already defines 31 signals, and we have already identified two new needed signals, SIGPARENT and SIGSCALL, in this paper as well as a new signal for checkpointing, SIGCKPT in [ELS88]. One solution to the problem is to extend the signal interface by adding new routines like **lsigsetmask** that take a pointer to a bitstring. Compatibility is only a minor issue because the extended interface is only required when the new signals are used. While this solution appears adequate to support the addition of a few new signal types, the problem seems general enough to be worth a more complete solution. We are considering the implementation of per-signal actions in a single language-specific first-stage signal handler instead of the kernel; this would allow a small number of maskable *levels*, but a nearly unbounded number of signal *types*.

6.3. Efficient Signal Masking

In uniprocessor operating systems, critical sections can be implemented by masking interrupts, but multiprocessors require some form of locking in addition. The same situation exists for serial and parallel applications, but with signals taking the place of hardware interrupts. When using signal masking and locking together, the overall performance depends on the efficiency of both mechanisms. Since we have identified low-overhead locking schemes, it remains to describe a complementary method of signal masking.

We propose to implement the signal masking functions without system calls, in a manner analogous to the implementation of temporary non-preemption described in section 3. In this case, the address of a single word (*sigword*) and two bitstrings in the user address space are maintained by the kernel. The user can set *sigword* to a non-zero value to mask all maskable signals, and *bitstring1* to mask only certain signals. The kernel examines *sigword* and *bitstring1*, and modifies *bitstring2* to notify the user of pending masked signals. The implementation of **sigblock** is essentially

```
sigword++;
oldmask = bitstring1;
bitstring1 = oldmask | newmask;
sigword--;
if (bitstring2 & ~bitstring1)
        sigcheck();
return oldmask;
```

where *newmask* is the argument and **sigcheck** is a new system call that causes all non-masked pending signals to be handled. The implementation of **sigsetmask** is analogous. Whenever a signal is sent to a process, the kernel examines *sigword* and *bitstring1* to see if it is masked or not, and it adjusts *bitstring2* if necessary whenever a signal is sent or handled. The use of *sigword* is not necessary on machines that support fetch&or, fetch&and, and fetch&store [Gottli81] on objects the size of the bitstrings.

7. Summary

We have presented a collection of proposals for enhancing the ability of UNIX systems to support sophisticated parallel applications on shared memory MIMD computers. It is possible in some cases to realize the advantages of an isolated subset of the proposed mechanisms, such as the scheduling controls discussed in section 3, but we believe that much of their strength comes from their use in combination, together with the memory management system of [ELS88]. Common themes include reliance on user-mode libraries and runtime support software for performance-critical functions, and extension of UNIX facilities to meet high-performance requirements.

Acknowledgements

The ideas described in this paper have benefited from many discussions with other members of the Ultracomputer Research Laboratory at NYU, and also with researchers on the IBM RP3 project.

References

[AT&T86] AT&T, *System V Interface Definition, Issue 2*, AT&T (1986).

[Convex87] Convex Computer Corporation, *Convex UNIX Programmer's Manual, Version 6.0*. Nov. 1, 1987.

[Cray86] Cray Research, Inc., *UNICOS System Calls Reference Manual (SR-2012)*. March, 1986.

[ELS88] Jan Edler, Jim Lipkis, and Edith Schonberg, "Memory Management in Symunix II: A Design for Large-Scale Shared Memory Multiprocessors," *Proc. USENIX Workshop on UNIX and Supercomputers*, (Sept. 1988).

[Fitzge86] Robert Fitzgerald and Richard F. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM Transactions on Computer Systems* 4(2) pp. 147-177 (May, 1986).

[Gottli87] Allan Gottlieb, "An Overview of the NYU Ultracomputer Project," pp. 25-95 in *Experimental Parallel Computing Architectures*, ed. Jack J. Dongarra, North Holland (1987).

[Gottli81] Allan Gottlieb and Clyde Kruskal, "Coordinating Parallel Processors: A Partial Unification," *ACM Computer Architecture News*, (Oct., 1981).

[Jordan78] Harry F. Jordan, "Special Purpose Architecture for Finite Element Analysis," *Proc. 1978 International Conference on Parallel Processing*, pp. 263-266 (August, 1978).

[Kepecs85] Jonathan Kepecs, "Lightweight Processes for UNIX Implementation and Applications," *Proc. Summer USENIX Conference*, pp. 299-308 (June, 1985).

[Ouster82] John K. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proc. 3rd International Conf. Distributed Computing Systems*, pp. 22-30 (1982).

[PBG85] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. 1985 International Conference on Parallel Processing*, pp. 764-771 (Aug. 1985).

[Reinha85] S. Reinhardt, "A Data-Flow Approach to Multitasking on CRAY X-MP Computers," *Proc. 10th ACM Symp. Operating Systems Principles.*, pp. 107-114 (Dec., 1985).

[Smith81] Burton J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," *Real Time Signal Processing IV, Proceedings of SPIE*, pp. 241-248 (1981). Reprinted in *Supercomputers: Design and Applications*, ed. Kai Hwang, IEEE Computer Society Press, 1984.

[Teixei88] Thomas J. Teixeira and Robert F. Gurwitz, "Stellix: UNIX for a Graphics Supercomputer," *Proc. Summer 1988 USENIX Conference*, pp. 321-330 (June, 1988).

[Tevani87] Avadis Tevanian, Jr., Richard F. Rashid, David B. Golob, David L. Black, Eric Cooper, and Michael W. Young, "MACH Threads and the UNIX Kernel: The Battle for Control," *Proc. USENIX Conf.*, (June, 1987).

[Thacke88] Charles P. Thacker, Lawrence C. Stewart, and Edwin H. Satterthwaite, Jr., "Firefly: A Multiprocessor Workstation," *IEEE Transactions on Computers* 37(8) pp. 909-920 (August, 1988).

[Zahorj88] John Zahorjan, Edward D. Lazowska, and Derek L. Eager, "Spinning Versus Blocking in Parallel Systems with Uncertainty," Technical Report 88-03-01, Dept. of Computer Science, University of Washington (March, 1988).

A Guest Facility for Unicos

Dennis M. Ritchie
AT&T Bell Laboratories
Murray Hill, NJ 07974

The COS system for Cray's X-MP series supports a guest facility, under which Unicos, their version of Unix® System V, can run as a subsystem in a fixed partition of the machine. The guest facility is unavailable when Unicos acts as a stand-alone system. This paper reports an experiment in which the author and David Slowinski added a limited version of the guest facility to Unicos, so that it can run itself as a subsystem.

COS's guest mechanism is analogous to that provided by other virtual-machine schemes, such as IBM's VM system, but is not as general. Cray supports it as a transition aid for sites converting to Unicos, and so far as I know, Unicos is the only guest system that has used the facility. As described below, Unicos is specially written to make it run as a guest, but the same binary system image runs both as a guest and in native, stand-alone mode. The main limitation in practice is that the machine is strictly partitioned—exactly one CPU is dedicated to guest Unicos, and the memory and disks are split statically. There is also some overhead in doing I/O in the guest system, but it is relatively small. The real penalty is the partitioning of main memory, and to a lesser extent the dedication of the CPU.

Providing one accepts its limitations, the guest facility is convenient for its intended purpose. At AT&T Bell Laboratories, for example, we ran guest Unicos on an X-MP/24 for 18 months before finally converting to the native Unicos environment. This path was followed for several reasons: we started with an early, not especially stable version of Unicos to which we made many changes, and control and use of the machine was shared between a company-wide computation center that sought stability and a research organization that was willing to experiment. Using the guest facility, existing COS applications could be moved gradually to Unicos, which benefited the computer center and its customers.

Our Unicos system developers, however, valued the guest facility mainly because it simplified testing. It was possible to reboot and try new versions of Unicos merely by submitting remote COS jobs, an operation readily automated; the process took a minute or two. Although doing this required chasing away the Unicos users, they were mostly from our own organization and arrangements could be informal. In particular, rebooting Unicos did not disturb the paying customers of the computer center.

Converting to native Unicos operation significantly impeded system development work; we were back in a world more normal for expensive machines, in which development takes place at most a couple of evenings per week, announced well in advance. Therefore, we decided to create a mechanism by which Unicos could support itself as a guest system.

How Unicos Works

The Cray hardware does not make it possible to create a true virtual machine in the style of IBM's VM system. That is, one cannot take an arbitrary standalone operating system and create a cocoon around it so that it believes it is in sole control of the machine. However, certain aspects of the hardware, and the style in which it is used, do ease the job of running Unicos as a guest. The great advantage is that most I/O operations are done through a separate processor, the IOS, and the CPU conventionally communicates with the IOS by sending messages to it. Thus I/O operations, which are often the most problematical aspects of a VM system, are readily handled if the communication with the IOS can be intercepted and the IOS operations simulated. Another complicated aspect of a virtual machine system, paging and virtual memory, is avoided simply because the hardware doesn't support it even for native systems.

Unicos, in particular, was already written to run as a guest. In native mode, I/O is accomplished by sending request packets to the IOS through a CPU channel. Guest Unicos instead leaves these same request packets in a conventional memory buffer and signals the host system by executing a trapping instruction. The device drivers are written in a style that prepares request packets and passes them to a low-level common routine; this routine is one of a handful of places that cares whether Unicos is running as guest or in native mode.

When guest Unicos needs assistance from its host system, it places a request code in a register, and executes a "normal exit" instruction, just as system calls are specified by user programs in a conventional operating system for the X-MP, whether COS or Unicos. The only real difference is that Unicos's requests on its own behalf are much simpler than the facilities it provides to its own users.

Thus the key to making Unicos run as a guest under itself is to build an interpreter for these requests. There seemed to be two approaches to doing this. One way was to add the interpreter to the Unicos kernel. In this model, a normal-exit trap executed by a distinguished guest process in simulated monitor mode receives special treatment. I/O packets extracted from a communications area are sent (after minimal processing) directly to the IOS or elsewhere. This is precisely the way COS handles things, and it is appropriate when performance is important. It is difficult to develop and debug a system based on this approach, however, because all the work is done in the system kernel. Instead, we decided on a scheme that is less direct, and less efficient, but moves the work of the simulator into an ordinary user-mode program and requires few kernel changes.

Kernel Changes

The two related kernel changes required to create our guest facility are modeled on the 'exchange' mechanism of the hardware. An exchange package, in Cray's terminology, is a data structure containing the most important user-visible registers and a collection of internal machine state variables, including the base and limit registers used for memory mapping. When the machine changes state, because of an internally-generated trap or an exogenous interrupt, the active exchange package is swapped with a new exchange package from memory. The usual operating system maintains an exchange package for each running user process, and one per processor for kernel execution.

A new system call, named *exguest*, takes as operands a hardware exchange package and a pointer to storage for machine registers not contained in the exchange package. The new call swaps the contents of the argument exchange package and registers with the system's copy of exchange information and registers for the calling process, and then begins execution using the new exchange package. In effect, it forces an extended version of the hardware exchange sequence, and is used to hand control from the simulator program to the simulated guest operating system.

Another change, needed to arrange proper return from *exguest*, affects the handling of exchanges into the kernel. Previously, these had always been treated by Unicos as device interrupts, or exceptions (like floating-point errors), or as Unicos system calls (normal exit traps). With the change, the system consults a new flag to determine whether the running process is under control of a guest operating system (that is, has executed the *exguest* call). If the flag is set, traps incurred are not treated normally, but instead generate a return from the simulator program's *exguest* call; during the return, the exchange information passed to *exguest* is changed to reflect the state of the hardware at the time of the trap.

Of course, this description is simplified. The exchange package passed to *exguest* must be checked for legitimacy, because it will ultimately be turned over to the hardware. The most important complication is that the base and limit values in the exchange package must be adjusted properly. When finally presented to the hardware, they must represent absolute machine addresses, while the addresses generated by the simulator are relative to itself. *Exguest* must arrange that these addresses are relocated properly, and that the locations available to the simulated operating system and its users fit within the address space of the simulator.

The Simulator Program

The simulator program runs as an ordinary user process. It does not need special permissions except that, as described below, it may be useful to allow it to read disk devices that most users cannot access directly. It begins by allocating a large amount of memory to serve as an arena in which to run the simulated operating system, and loading the operating system code into this space. The main loop of the simulator repeatedly hands control to the guest system with *exguest;* on return, it examines the resulting exchange package to see what is being requested. Requests fall into two classes: those from the simulated operating system itself, which either map into appropriate I/O operations or ask to run a user program, and traps incurred by user-mode programs running inside the guest operating system. These are treated simply by forwarding control to the guest operating system.

More explicitly, the flow of control in the simulator program can be illustrated by the following pseudo-code:

```
forever {
    exguest(kernelXP)
    if kernelXP.requestreg == IOREQUEST
        doIO
    else if kernelXP.requestreg == RUNUSER
        exguest(kernelXP.paramreg)
    else
```

```
        panic
    }
```

That is: exchange to the guest kernel; when it exchanges back, it is either requesting an I/O operation, or running a user program. In the former case, it suffices to do the needed I/O and continue. In the latter, the request carries a parameter specifying the desired user exchange package. The second *exguest* call in the example exchanges control to this simulated user process, and returns when the process makes a system call or incurs some other trap. The user exchange package, which resides in the simulated kernel's address space, is updated by the exchange, so it suffices merely to continue; the guest kernel will process the user's system call or trap.

The guest kernel handles I/O operations by bundling parameters into a queue in a conventional spot, and the *doIO* routine pulls requests out, does the transmission, and updates a reply queue. The requests are elementary; they specify a device, an address within the device, the transmission direction, a memory address, and a count. In order to make the simulator useful at all, the devices and their addresses have to be mapped somehow into a real file system on a disk or SSD. Therefore, the simulator contains a table indexed by device and sector address range that yields a Unicos device name and offset. A side table contains file descriptors for devices that have previously been used by the simulator. When the guest system refers to a sector on a particular device, the main table is consulted to map the request into an address within the corresponding Unicos device file; ordinary Unicos I/O calls are used to read or write the requested information, after opening the device if necessary.

By setting up the simulator tables appropriately, it is possible to control access to all the disk and SSD storage on the machine, and in particular for the simulator to see the devices on which user's files are kept. However, the simulator accesses file systems directly, through the raw disk device, and thus bypasses Unicos's cacheing strategy. The simulator should not be allowed to write a disk partition that is also mounted as an ordinary file system. One simple way to manage things is to create a root partition, containing a near-copy of the real root, for the simulator's exclusive use. A few files in the simulator's root file system are modified; for example, the script that mounts new file systems after startup mounts user file systems read-only, so that the simulator never attempts to change them.

Asynchronous events

The scheme described above for I/O works well for operations on the disk and SSD that complete quickly and can be made synchronous. Some kinds of I/O, and other things, occur asynchronously and must be handled by other means.

Console output, which is specified by a special kind of IOS packet, is straightforward, but input occurs asynchronously, and waiting for it must not block the simulation. Therefore, the simulator loop above must be supplemented by code that checks its standard input for characters every now and then. When it finds any, it generates an appropriate packet, and inserts it into the packet-input queue of the guest system. Similarly, Unicos expects to receive clock interrupts periodically. Therefore, each time around its loop, the simulator checks the real-time

clock, and if it has advanced sufficiently, simulates a clock interrupt by setting the appropriate bit in the kernel exchange package. Fortunately, Unicos is written in such a way that it is insensitive to wide variations in the frequency of clock interrupts, so there is no need to be concerned about timing them precisely.

Limitations

The current simulator is a crude program, and is not at all suitable as a production tool, as the COS guest facility is. It is useful for system development.

There is an unavoidable cost in using our approach, in that I/O operations in the guest system turn into ordinary Unicos I/O calls in the simulator. This means they are much slower than they would be if the simulator were integrated into the operating system itself.

Moreover, the only I/O devices we support are the console (and only one console), the disks, and the SSD. In our installation, we use a low-speed CPU channel to interface to our Datakit® network, and this channel is not included in the simulation. The problem here is not the simulator itself, as much as is the interface to Datakit presented by the operating system. It appears to user programs as a set of devices each referring to a single virtual circuit on the network. However, the low-level interface to the hardware involves messages specifying the circuit number. The operating system does not provide a raw Datakit device in same way it does for disk devices. Equally important, Datakit circuit setup involves communication with a network controller, and the controller software does not provide a way of managing two independent sets of connections (host and guest) over the same physical channel.

A process running as a guest operating system is an ordinary Unicos process, and necessarily is large, because it must contain not only the operating system code but also enough memory for its own user programs. While it is active, it swaps against the rest of the system load. Moreover, the simulator (as written) has two unappetizing choices in managing its CPU usage. Either it runs all the time, faithfully simulating the idle loop, or it detects that the guest system is idle and waits a bit. Unfortunately, the obvious ways of waiting encourage the real Unicos system to swap the simulator out, or otherwise delay it. The former burns one whole CPU, the latter makes it quite slow. Of course, this situation resembles the one we accepted while we were running guest Unicos in production, but now the production programs use more memory, and the accounting statistics show the simulator CPU usage explicitly. Nevertheless, the simulator load is quite perceptable to the users, and they are not always assuaged by assuring them that a year or so ago they would not have been able to run their programs at all, or that the alterative is dedicated development time.

Summary

Our intent in creating a Unicos guest facility was to make it possible to do system development without disturbing production users, not to build a a mechanism that itself would permit running production codes. Even so, there remain many loose ends. For example, as discussed above, our networking hardware is not supported. Nevertheless, it is useful for its intended purpose, and it is much easier to check the integrity of new system code under the guest

mechanism than to arrange for standalone development time. Also, the scale of the project was small. The user-level simulator program contains only about 500 lines of C code, and the operating system additions total less than 200. The effort seems well repaid.

Distributed Supercomputer Graphics Using UNIX™ Tools

H. Stephen Anderson
The Ohio Supercomputer Graphics Project
The Ohio Supercomputer Center
1224 Kinnear Rd.
Columbus, Ohio 43212
(614) 292-3416

ABSTRACT

This paper describes an environment for *distributed visualization* over a network that includes a Convex C-1, Cray X-MP/24, and numerous workstations, all running UNIX*. The presence of UNIX on supercomputers allows for a homogeneous software environment across the network. This consistency, combined with some of UNIX's powerful concepts, yields a solid base for the construction of a data flow model for distributed graphics. This environment promotes *appropriate* use of computing resources; each machine in the local network is used for the purposes it is best suited, relieving the supercomputer of some of the burden imposed by the traditional batch method of supercomputer operation. It also promotes clean, portable code which speeds up the software development cycle, saving computing resources and human resources.

UNIX concepts such as pipes and remote shell execution are described in this context. Performance is discussed showing that this model uses fewer total resources, consuming fewer supercomputer CPU cycles and supercomputer disk accesses without increasing the load on the network. This model also allows the supercomputer to be included in the *graphical network*, a key step toward developing an environment for scientific visualization.

A case study is presented representing work in progress at The Ohio Supercomputer Graphics Project, part of The Ohio Supercomputer Center. The data flow environment has provided both good tools to build application software for visualization and a good testbed for future software research.

1.0 Introduction

UNIX is the defacto standard operating system for workstations, the most popular operating system for graphics applications, and is beginning to be widely accepted in the sciences. It is only natural that UNIX should emerge on supercomputers. The next logical step is to integrate the supercomputer into the computing network.

The presence of UNIX on supercomputers allows for a homogeneous software environment across the computer network. The benefits of having the same operating system on the supercomputer as the local workstation, and having that operating system be UNIX are tremendous: familiar user and software development environments, broad base of programs available, and powerful network connections. Benefits of familiar user environments and a broad software base are obvious. Because UNIX is excels at connecting to other computers running UNIX, simple UNIX concepts can be used to integrate a supercomputer into the computing network.

---

* UNIX is a registered trademark of AT&T Bell Laboratories

The application discussed in this paper is an environment for scientific visualization which features unidirectional connections between modules based on UNIX primitives. None of the ideas presented are unique to graphics; any data-flow type application could be substituted. Previous work in graphics using a similar approach can be found in work by Potmesil and Hoffert[4] and Nadas and Fournier [6]. Another application of data flow approaches to computer graphics is described in a paper by Hedelman [5].

This work was conducted as part of a larger effort at The Ohio Supercomputer Graphics Project (OSGP) at The Ohio Supercomputer Center. The Ohio Supercomputer Center (OSC), is a statewide supercomputer facility, made up of a consortium of universities and located on the campus of The Ohio State University. The center serves remote users over a state-wide network, and features exceptional facilities for producing graphics and animation at the center. One of the major goals of OSGP is **apE**, the **a**nimation **p**roduction Environment, a flexible, integrated, object-oriented environment for graphics; in particular scientific visualization. The work described in this paper (nicknamed **chimp**) grew from the desire to support users of the Ohio Supercomputer Center immediately while further research in visualization is conducted.

1.1 Distributed Visualization

Scientific visualization has become a popular buzzword in the sciences and computer industry lately. The idea of wanting to create visual output from a scientific simulation is not new, but only recently have people begun to investigate the process of visualization and think about developing software with scientific visualization in mind. Simply stated, visualization is the process of aiding data analysis and comprehension by transforming the symbolic (numbers) into the geometric (images). For a deeper background on scientific visualization [1] is the definitive reference.

One of the first problems that confronts anyone attempting scientific visualization is that computers designed for interacting with users and producing visual information (workstations) do not typically have the processing horsepower to perform the computational simulation. Computers designed for performing the complex computational simulation (supercomputers) do not typically have methods of easily bringing the visual results of the simulation to the desk of the scientist. One approach to address this problem is to use *distributed computing*; having different computers on the network perform pieces of the larger problem. During execution the data flows between each program module, any of which may reside on a remote machine.

The process of transforming the scientist's simulation into an image or animation can be thought of as a pipeline. The data flows through the set of filter programs, creating the final image:

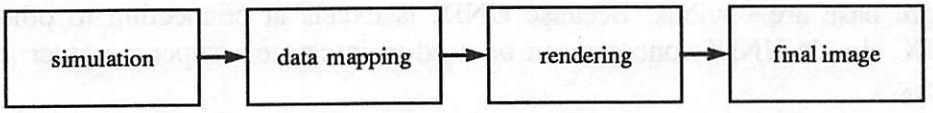


figure 1: visualization pipeline

Note that in this simplified model, all data paths between pieces go in only one direction. This is an assumption which ultimately will limit the flexibility and interactivity of the system, however, this model is still powerful enough to construct a visualization system with the desired properties.

The steps outlined in figure 1 can be broken up and distributed among different machines connected by the network: The local workstation is used to set up the simulation, monitor progress of the simulation, and view the final output. The supercomputer is used primarily for the intense calculations of the simulation model, and perhaps mapping from simulation into image space. Ideally, the overhead to set up such a simulation should be minimal and the inter-computer connections should be fairly transparent.

Parallel to the idea of distributed visualization is the concept of a *graphical network*. A successful visualization environment should distribute the visual data to the desks of the users with the same fervor that the numeric data has been distributed over the ethernet. Forcing users to leave their desk and walk down the hall to view their graphical simulation is only slightly less objectionable to forcing them to wait until the next morning to retrieve a stack of output printed by the line printer.

1.2 Appropriate Use of Computing Resources

Distributed computer systems offer four main advantages over traditional serial computing models [2]: concurrent processing, load balancing, improved reliability, and resource sharing. Although all of these benefits are important for our application, our main interests are with load balancing and resource sharing. Moving any computations from the supercomputer to the local workstation is greatly appreciated by other users who are competing for supercomputer resources. Using the local computing power to it's potential and using the supercomputer only when necessary helps balance the total network computing load.

Distributed processing allows the supercomputer to share the local display devices of the workstation. Visual output from the supercomputer simulation can be directed to the display of the local workstation or a frame buffer on the local network. Playback of an animation sequence can be accomplished by collecting the frames for recording to videotape or by direct digital playback in the memory of the workstation.

By using the supercomputer in a more interactive mode, computations can be monitored during execution so that erroneous calculations can be stopped. In a batch environment it is not at all unusual to submit a job and wait for the results, only to find that an error has causes the output to be meaningless. This process wastes supercomputer CPU cycles and valuable user/programmer time. The ability to monitor simulations while they are happening allows the user to prevent some of this waste.

2.0 chimp Overview

This section gives an abstract overview of the **chimp** system. **chimp** is built on top of the UNIX operating system and uses UNIX features familiar to any intermediate-level UNIX user. Programs are one of two types: *tools* or *filters*. A tool is a window-based interactive program used to examine or manipulate data or construct a data flow description. A filter is a tool which reads in data in a data flow description, possibly manipulates it in some way, and

writes it out in a data flow description. An *application* is a combination of tools and filters which performs some useful function. Applications are typically written as UNIX shell programs or small C routines.

The data flow format is attribute based, each field is identified by a name and its value(s). Information in the data stream may be parameters to be interpreted by a filter or tool (a flow description), or data. All flow descriptions are transmitted as ASCII characters, allowing easy reading, editing, or generation of flow descriptions. Data may be either ASCII or binary. Routines to convert from ASCII to binary and vice versa are provided as filters. Binary data is communicated in a machine independent format, which allows it to be transmitted between machines with different binary formats.

```
begin normalize
   type   linear
   range  0.01664 0.68485
   symbol "energy"
end normalize
```

figure 2a: data flow created by the histogram tool

```
begin map "This is a mapping test"
    note "density from Comer Duncan"
    type MapHSV
    grid_name  "RectBlackHole"
    begin mapping MapHSV_H
       resolution  32
       symbol  "density"
    end mapping
    begin mapping MapHSV_S
       symbol  "energy"
    end mapping
end map
```

figure 2b: data flow to be interpreted by the color mapping tool.

```
begin grid "BlackHole"
   type    uniform
   system  polar
   dimensions 2
   begin axis "Radius"
      spacing irregular
      size 150
      origin 0.483333
      length 4.966667
      position[150]
         0
         0.0333333
         0.0666667
         0.1
         ...
   end axis
   symbols[2]
      "energy"
      "density"
end grid
```

figure 2c: data flow created by the simulation

figure 2: Sample data flows

Each filter or tool operates on one or more flow descriptions, and possibly some data. The **chimp** system is loosely coupled, promoting flexibility. Flow descriptions may be created with a tool, a user program, or the user can use a text editor to create one by hand. The connection between the scientific simulation and the **chimp** system is made by the scientist. As part of his simulation, he constructs and outputs a basic flow description, which simply involves dumping his data structures. He then uses the **chimp** tools to interactively define any viewing parameters, color mappings, or data normalizations necessary to produce the desired images. Using these flow descriptions, an animation can be created by running all of the data through filter programs.

All **chimp** programs use standard UNIX input and output schemes, reading input from *stdin* and writing output to *stdout*. UNIX provides commands on the system level to direct input and output from any program, without that program's knowledge. Input can be directed to come from the keyboard, or alternatively from a file. Output can be directed to the screen for examination, or to a file to be saved.

2.1 Pipes

A pipe is a UNIX construct which allows the output of one program to be connected as the input of another. UNIX automatically handles all of the buffering of the data in the pipe. When a program attempts to write to a pipe that is full, the program will block until some data is read. Likewise, a program reading from a pipe will block when there is no data to be read.

Pipes are used in our system for communication between tools and/or filters of an application. Data and program parameters flow through the system in the data flow format. Commands or data which have no meaning to a particular program are propagated down the pipeline.

2.2 Remote Shells

An obvious extension to the previous discussion extends the pipe syntax across the local network, allowing individual building block processes to be distributed on different machines. UNIX provides a simple way to do this with the *rsh* (or *remsh* in System V) syntax.

rsh runs a shell on a remote machine, connecting `stdin` and `stdout` as appropriate with the local shell invocation. In this way, pipelines can be constructed using both local and remote processes:

```
cat start.data | rsh convex norm | rsh cray resamp | color sunv8
```

This command would load the pipeline up with the contents of the file `start.data`, piping it to the program `norm` on the machine named `convex`. The output of `norm` would be connected as input to the program `resamp` on the `cray`, the output of which would be connected as input to the program `color` back on the local workstation. This would have the effect of normalizing the data, resampling the data grid, and colorizing the grid, creating a two-dimensional color image on the local display.

Depending on the volume of data, and the execution speed of the program modules, executing some modules remotely could decrease the execution time for the entire pipe. Another seemingly obvious drawback is the load on the network, but the bandwidth is the same as copying the data files between the local and remote machines. The only way to actually lower the bandwidth is to not use the remote machine, but this would either sacrifice all the computational advantages of the supercomputer, or sacrifice the ability to display the graphical results on the workstation.

A major advantage of this method is the decrease in disk access on the supercomputer. Since the supercomputer only reads and writes to the network, reads or writes to the disk of the supercomputer are not done.

3.0 Examples

The following sections illustrate use of the **chimp** system.

3.1 Scientific Visualization

The **chimp** system has been used to generate images from scientific simulations. Let us illustrate with an example: Dr. G. Comer Duncan, an astrophysicist from Bowling Green State University has developed a computational model of a star colliding with a black hole at various velocities [7]. His simulation, running on OSC's Cray X-MP, produced a data flow description of his sample space and variable values. Using interactive **chimp** tools, he created data normalization definitions and a color mapping definition. A UNIX shell command which would send the data through the pipeline and generate an image is:

```
cat enernorm.flow resamp.flow map.flow grid.flow  var.flow |
     rsh cray norm | rsh convex resamp | color -V sunv8
```

`grid.flow` and `var.flow` were created by the simulation. The other flow definition files were created with the tools. The ungainly command above was encapsulated into a shell script which generated several hundred frames of an animation, one frame of which is shown in figure 3.

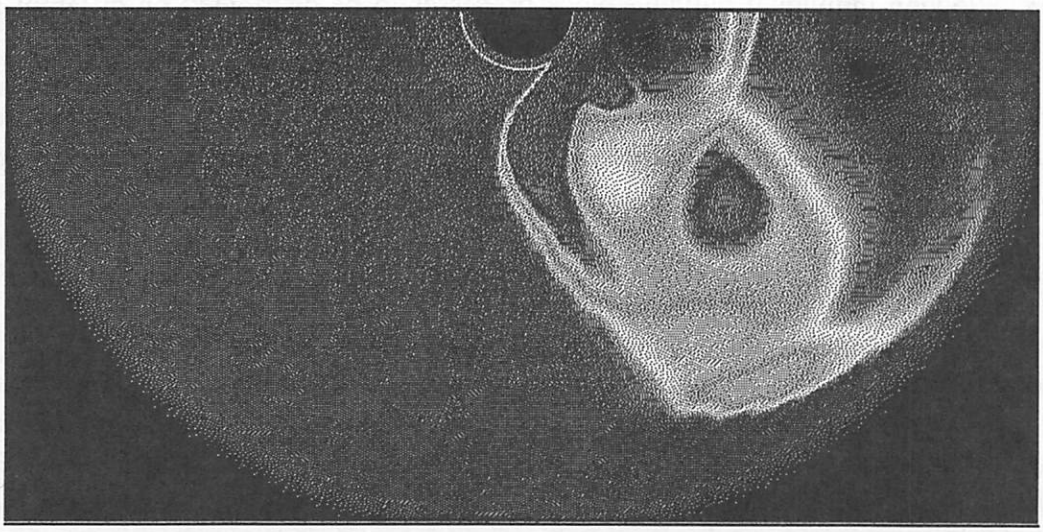


figure 3: Collision of a star with a black hole, courtesy of
Dr. G. Comer Duncan, Department of Astronomy and Physics,
Bowling Green State University [7].

3.2 Supercomputer Rendering

Another example of applying these concepts is rendering computer animation on supercomputers. All data modeling, animation scripting, etc., can be done on an interactive workstation. The compute-intensive rendering stage can be off-loaded onto a supercomputer, with the image results directed to a local frame buffer for viewing or recording (**figure 4**).

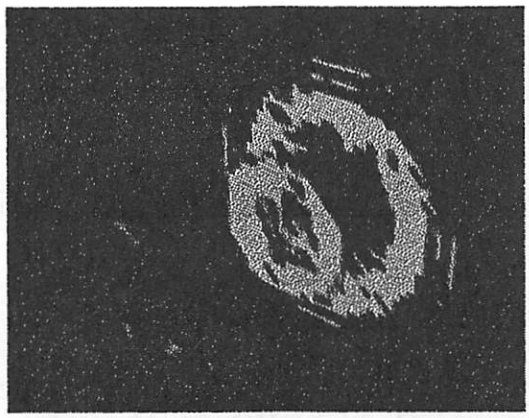


figure 4: Polygonal rendering of the relative humidity of a hurricane, courtesy Dr. Jay Hobgood, Department of Geography, The Ohio State University **[8]**.

4.0 Conclusions

UNIX is good choice of operating systems for supercomputer class machines because it helps integrate the supercomputer into the computing network. Providing a homogeneous software environment across the network creates many benefits: familiar user environments, broader base of available software, and powerful network connections. The ability of UNIX systems to connect to other UNIX systems allows for the simple construction of a distributed graphics environment.

Simple UNIX constructs can be used to create a flexible environment which is easy to use, promotes appropriate use of resources, and capable of taking advantage of distributed processing. Developing additional software within the **chimp** system is straightforward. There are some limitations to the system (most notably the dependency on UNIX pipes for inter-module communication), but use of **chimp** by scientists has shown that it can provide needed visualization tools.

4.1 Future Work

Future work includes extending the capabilities of **chimp** by writing more tools and filters and providing a window-based tool to allow interactive construction of pipelines. Removing the dependency on the pipeline structure, and providing for asychronous bi-directional communication will allow for interactive steering of the simulation.

Acknowledgments

This paper represents the work of many people at The Ohio Supercomputer Graphics Project, most notably: John Berton Jr., Peter G. Carswell, Scott Dyer, Jeff Faust, and Bob Marshall. Thanks also to Naomi Josephson for editing the paper. The work in this paper and all the work of The Ohio Supercomputer Graphics Project could not be accomplished without the support and creation of the stimulating environment by Professor Charles Csuri, Dr. Charles Bender, and the users of The Ohio Supercomputer Center.

References

[1] McCormick, B. H, et al. (ed), "Visualization in Scientific Computing", *Computer Graphics,* Volume 21, No. 6, November 1987.

[2] Peterson, J., Silberschatz, A., Operating Systems Concepts, Addison-Wesley Publishing, 1983, pp. 437-439.

[3] Rochkind, M., Advanced UNIX Programming, Prentice-Hall Inc., 1985

[4] Potmesil, M., Hoffert, E., "FRAMES: Software Tools for Modeling, Rendering and Animation of 3D Scenes", *Computer Graphics,* Vol. 21, No. 4, July, 1987. Proceedings of SIGGRAPH 1987, pp. 85-93.

[5] Hedelman, H., "A Data Flow Approach to Procedural Modeling", *IEEE Computer Graphics and Applications,* January 1984, pp. 16-26.

[6] Nadas, T., Fournier, A., "GRAPE: An Environment to Build Display Processes", *Computer Graphics,* Vol. 21, No. 4, July, 1987. Proceedings of SIGGRAPH 1987, pp. 75-84.

[7] Duncan, G. C., "Head-on Collision of a Black Hole and a Star", to appear in the proceedings of the Fourth Science and Engineering Symposium, sponsored by Cray Research Inc., October 1988.

[8] Hobgood, J. S., "A Possible Mechanism for the Diurnal Oscillation of Tropical Cyclones", Journal of Atmospheric Sciences, Vol. 43, 1986, pp. 2901-2922.

The CTSS/POSIX Project

Jonathan Brown

P.O. Box 5509, L-560
Lawrence Livermore National Laboratory
Livermore, CA 94550
(415) 423-4157
jbrown@nmfecc.arpa

ABSTRACT

The IEEE has produced POSIX, a proposed standard portable operating system based on UNIX. It has been adopted as a draft American National Standard (ANSI), a proposed Federal Information Processing Standard (FIPS), and has been endorsed by most major computer manufacturers. To profit from this new common development environment, the NMFECC is integrating CTSS, its supercomputing operating system, with a standard UNIX compatible environment. The CTSS file structure has been extended to be compatible with UNIX. Over one hundred UNIX utilities have been ported to the UNIX environment in CTSS. Work is in progress in the operating system, library, and utility software to achieve full POSIX compatibility. Upon completion, the NMFECC will provide a standard POSIX conforming UNIX environment in addition to the current CTSS environment. The new UNIX environment will profit from the security, checkpoint recovery, high-performance I/O, vectorizing/optimizing FORTRAN compilers, multitasking, and other supercomputing features native to CTSS. CTSS will benefit from the rich set of software available in the UNIX world. The resulting hybrid system will combine the complementary strengths of both UNIX and CTSS.

Real Productivity for Real Science Without Real UNIX

Robert M. Panoff

Department of Physics and Astronomy
Kinard Laboratory of Physics
Clemson University
Clemson, SC 29634-1911

ABSTRACT

A real fear of many active researchers is that the advent of UNIX to the supercomputer environment will significantly impair their productivity rather than enhance it. Three stumbling blocks which present themselves are poor compiler technology, inadequate debugging tools, and process proliferation. Using Quantum Monte Carlo calculations as a paradigm, I will discuss how highly productive research without UNIX has been accomplished, suggesting that nothing unique to the UNIX environment would facilitate this work significantly. For whatever reasons UNIX is being promoted as a primary operating system of the future, increased productivity of current supercomputer users appears doubtfully among them.

Numerical Applications Interprocess Communication Protocol: RPCODE: RPC server to solve ODEs

C. A. Stewart

Open Systems Architects, Inc.

"Implementors should think of the RPC protocol as the jump-subroutine (JSR) of a network"[1]

ABSTRACT

Described here is a pilot project to specify a robust interprocess communication protocol at the applications layer for the distribution of numerical program segments to various compute servers on a local area network. The model program is a Sun RPC (c) server to calculate the solutions to several sets of ordinary differential equations to run simultaneously on a number of Sun-3 workstations, a Cray-2 and a Gould 9060; and to report back to a client program running on a single Sun-3 workstation. Some illustrative implementation details are presented. The use of connected vs. connectionless transmission, the management of parallel program execution on a variety of hosts through a single client calling up several servers, and the implications of developing a more complete numerical algorithms communications protocol are discussed.

1. Introduction

In today's heterogeneous networked computer environment one of the most perplexing issues is application/host level load balancing: how to configure a number of different kinds of computers on a network to best serve a changing suite of applications--and how to configure specific applications to better utilize a particular network of compute servers and workstations. Ideally, each host would handle an application segment suited to that host's architecture, operating system and special capabilities. In numerical scientific computing, the application segments are typically calls to numerical routines. Ideally, a group of supercomputers on a network could exchange numerical instructions and data among eachother. Ideally, there would be a set of independent development tools to evaluate the and reschedule the mapping of numerical routines to hosts, such that a range of scheduling algorithms and locally developed scheduling heuristics may be tested.

Numerical routines are typically linked to a program that runs entirely on a single host. The most advanced computation laboratories have a network of high-speed computers, among them architectures as diverse as the Intel Hypercube, the Alliant FX8, the ETA-10, the Cray-2 and IBM 3090, all of which do or are capable of running some flavor of UNIX®. These represent a range of memory sizes, vector

[1] RFC 1057

and scalar processing speeds as well as a range of parallel granularities and examples of MIMD and shared memory parallel architectures. While there are numerous research and development efforts aimed at improving specific numerical algorithms[2] and aimed at scheduling tasks[3] to the processors on architectures real[4] and imagined, it is also desirable to have an application-level language to call those routines over the network, so that each routine is most easily accessible on the machine best suited for that numerical task.

The functional requirements of such a language/data standard are defined by the entry points and data requirements of numerical software libraries. Considered here are ordinary differential equation solvers from the IMSL and NETLIB libraries. Sun Microsystems' eXternal Data Representation (XDR) and Remote Procedure Call (RPC) are the appropriate tools for building machine-independent data structures and network-based communications with remote routines for UNIX (TM) supercomputers.

2. Implementation

Combining standard numerical routines with Sun RPC and XDR starts with the benefits and drawbacks of mixed language programming. This is a subtly documented feature of UNIX (TM) systems, where C and Fortran share an object code format, and can access eachother's entry points if properly enticed. Passing addresses rather than values to the fortran functions is just the beginning. On the Gould, it is the FORT fortran compiler, not f77 that is accessible from C, and the main routine must be called "MAIN_", not "main". On the Cray 2, CFT not f77 is the necessary compiler, which is not well integrated with 'make' requiring kluges such as:

```
.SUFFIXES: .F .f .o

.f.o:
    cft $(*).f

.F.o:
    /lib/cpp $(*).F | sed '/#/d' > $(*).f
    cft $(*).f
```

The benefits of mixed-language programming are that the numerical routines do not need to be re-written, they can be compiled to take full advantage of vectorizing fortran compiliers, and they can access arrays that have been dynamically allocated by the C RPC routines.

The second implementation problem to be solved is to provide the numerical routine with all the information it needs to solve the problem. One of the drawbacks of RPC network servers is that they do not share a common data space in memory with the calling client. This means that if the numerical routine typically shares information with the calling routine by means of common blocks, those common blocks must be constructed by the server, with information passed to it in the XDR format arguments of the calling client routine. The ODE solvers implemented in this project (the IMSL routine dverk and the ODEPACK routine lsoda) do not make extensive use of common blocks, but require them implicitly if the values of the coefficients of the ODEs are to be changed with each new run.

In addition to usual procedure arguments and common blocks, ODE solvers require the specification of an external routine specifying the functional form of the ODEs and, for stiff ODEs, it is desirable to provide an additional external routine which specifies the Jacobian matrix. The structure of these requirements do reflect some of the limitations of Fortran as a language: for instance, the functional form of the Jacobian is a simple logical inference from the functional form of the ODEs, so to specify the functional form of the Jacobian in a second external routine is to provide redundant information. In this study, this has been done by hand: we do not have a *general* network-based ODE service, but rather a service which has options to solve several example sets of ODEs (one being the equations of Lorenz which produces the famous "strange attractor" for certain parameter values and initial conditions), allowing the manipulation of the ODE coefficients through common blocks reconstructed on the server.

To this effect, the following C fragment from the server:

```
SPARAMS (&in.param[0], &in.param[1], &in.param[3]);
for (i=0; i < in.ntimes; i++) {
    t = i*in.tstep;
    tnext = (i+1)*in.tstep;
    DSALT (y,&t,&tnext,&in.tol,&ind,&ier);
}
```

calls the following Fortran routines:

```
subroutine sparams (r, a, n, e1, e2)
common/terms/tcs(12),tcc(12,4),itc(12,4,2),ntc(12),ec1,ec2
dimension ucc(8),wcc(8)
    ...
wcc(7) = -(r*(a*n)**2)/(((a*n)**2 + (2*zpi)**2) **2)
    ...
return
end
```

where "..." means more of the same.

```
subroutine dsalt (y,t,tnext,tol,ind,ier)
dimension y(12), w(12,9), c(24)
external cheryl
n=12
call dverk (n,salt,t,y,tnext,tol,ind,c,n,w,ier)
return
end
```

Sparams constructs the common block for the ODE coupling coefficients for Saltzman's equations, the ODE functional form:

```
subroutine salt(n,t,tc,tcprime)
dimension tcprime(12), tc(12)
common/terms/tcs(12),tcc(12,4),itc(12,4,2),ntc(12),ec1,ec2
do 700 i = 1,12
    tcprime(i) = tcs(i)*tc(i)
    do 700 j  = 1,ntc(i)
        tcprime(i) = tcprime(i) + tcc(i,j)*tc(itc(i,j,1))*tc(itc(i,j,2))
```

```
700   continue
      tcprime(9) = tcprime (9)  + ec1
      tcprime(11) = tcprime (11) + ec2
      return
      end
```

The advantage of calling the wrapper fortran routine DSALT which calls the ODE solver, rather than having the function salt called from the C function is that it avoids inconsistencies in C/Fortran character-handling between installations.

For ODEs with constant coefficients, it is desirable to construct the common block in a separate routine prior to numerical integration when many coefficients for the ODEs are derived from a few parameters, as in sparams. ODEs with non-constant coefficients have a choice between calculating their coefficients inside the function that specifies the functional form of the ODEs or outside; one is more accurate in a routine that subdivides its time-steps, the other is more efficient and appropriate where the characteristic time-scale of the nonautonomous terms is longer than the time-scale associated with the linear or nonlinear couplings among the equations in the parameter range of interest. It is unfortunate that these are decisions that must be made at compile time. It would be desirable for the client to pass the functional form of the ODE to the server, and for the server to determine the jacobian automatically, determine whether and when the problem is stiff, write and compile these functions automatically, and allow for their interactive correction, modification and more detailed study of the integration process itself through a remote debugger. Parsing ODE solvers are available for the IBM PC[5] and Macintosh[6] which make extensive use of assembly language and the simplicity of the systems, but are limited in the size of the problems they can handle by the same. The aim here is to establish that functionality in a portable and efficient manner for access as services on a large network of workstations and supercomputers.

It is also important to keep in mind that a single set of ODEs can interest large numbers of people for many decades.[7 8 9 10] Therefore, an ODE-parsing server can reasonably start with certain systems already installed. When set up to serve many users, there needs to be some distinction between permanently installed systems and an individual user's development system. There is the need to load the same set of ODEs on a large number of servers simultaneously, since parameter-space and initial condition sweeps can be quite useful in studying the more global behavior of these systems. In this version, TCP was used rather than UDP because of the potential for large amounts of data being transmitted and the immediate desire for a reliable connection. This was necessary for the Saltzman equations but not absolutely for the Lorenz equations. There are circumstances where the connectionless mode is more useful: when the number, complexity and stiffness of the equations is low (e.g. the forced Van der Pol equations) and the characteristic timescale of interest is relatively short, the point is to get as many machines working at once on different parts of initial condition/parameter space the client should go into broadcast mode, handling timeouts and retransmits as necessary.

3. Performance

RFC 1057 claims that a network server should take one *or more* orders magnitude longer than a locally linked function. However, the actual figures for more

CPU-intensive servers are significantly different. One fair comparison is to eliminate disk I/O or format conversion of data completely, and to compare the total elapsed time of a program linked directly to the ODE solver on a Sun 3/260 plus floating-point accelerator (fpa) to the total elapsed time of the same call made over the network from a Sun 3/50 which is not a client of the 3/260. The locally linked procedure runs four times faster than the same fpa-optimized routine called over the network as a service. When the locally linked procedure is called from the 3/50 using a remote shell, it is approximately three times as fast as the RPC server. When the locally linked procedure is called from the 3/50 using 'on' and 'rex' the performance is the same as that called using a remote shell. When the locally linked process is started via a fast rlogin, and the login shell overhead is counted, *the total elapsed time is the same.*

The fact that the process and network overhead incurred by calling a remote routine *can* be an order of magnitude or more greater than the overhead incurred by calling a local routine simplifies the task of specifying an appropriate language: only the highest-level numerical routines should be accessed as RPC services. For instance, the BLAS (Basic Linear Algebra Subroutine) routines to do a scalar add to a vector are clearly not suited for repeated access over the network: performance is necessarily degraded relative to a local call due to network and process overhead. For the procedures that repeatedly call BLAS locally (e.g. LSODI[11]), the ratio between local and network service performance of the higher level routine improves as the network service becomes more CPU bound rather than I/O bound.

4. Further Development

It's not clear that the benefits of the mixed-language approach outweigh its drawbacks. One, vectorizing C compilers are becoming available. Two, on many machines (e.g. the IPSC), the more natural language for writing code of any kind is C. Three, the passing of information through common blocks in Fortran essentially requires additional function-call overhead on both the client and server side to bundle and unbundle the common blocks as well as perform conversion to a host-independent data format. Four, the performance, although far better than expected, is not enough to justify the amount of work involved with installing a new functional form of the ODEs under the Fortran language. It is therefore concluded from this study that the parsing, compilation, debugging and installation of new functional forms of the ODEs would be made simpler by a rewrite of the existing routines in a more flexible language. These conclusions are, to a certain extent, peculiar to the solution of ODEs and also minimization problems, because the specification of a user-specified functional form within the limitations of the Fortran language requires compilation in the server.

5. Discussion

The goal of this project is to create a few flexible network-based numerical computing tools for users and developers. The adoption of a suite of RPC-based numerical software services as the primary means of supercomputer use could alter patterns of supercomputer use. Network traffic would increase, but small user programs that generate numerous interrupts (e.g. vi) would decrease. user-generated data I/O patterns would be more easily managed: unpredictable small reads and writes to local disk would be replaced by streams or datagrams to the network, and become the workstation owner's problem. A change in process and

I/O load patterns could simplify the optimization of the scheduler code on a single host. Furthermore, load-balancing among applications and hosts would be more amenable to network administration: the mapping of primary numerics services to hosts could be interactively or automatically updated depending on use, and locally applicable heuristics could be built in to the network-level scheduling. Furthermore, the mechanisms by which each service is evaluated during development and testing — resource utilization records — could be easily adapted for accounting purposes. It may be desirable, with the implementation of accounting and authentication features, to offer RPC numerical services to a wider variety of users on the Internet on a subscription basis, with fewer login accounts to be maintained by a supercomputer organization.

References

[1] Sun Microsystems, *Request for Comment #1057*, Network Working Group, June 1988.

[2] Jorge R. Leis and Mark A Kramer, The Simultaneous Solution and Sensitivity Analysis of Systems Described by Ordinary Differential Equations, *ACM Transactions on Mathematical Software*, **14** (45-60) 1988.

[3] Shlomit S. Pinter and Yaron Wolfstahl, On Mapping Processes to Processors in Distributed Systems, *International Journal of Parallel Programming*, **16** (1-15), 1988.

[4] C. L. Seitz, The Cosmic Cube *Communications of the ACM* 28 (22-33), 1985.

[5] Huseyin Kocack, *Differential and Difference Equations therough Computer Epxeriments*, Springer-Verlag 1986.

[6] John Hubbard, *Unpublished Draft*, 1988

[7] E. N. Lorenz, Deterministic Non-periodic Flow, *Journal of the Atmospheric Sciences*,**20** (130-141) 1963.

[8] Colin Sparrow, *The Lorenz Equations: Bifurcations, Chaos, and Strange Attractors*, Springer-Verlag, 1982.

[9] Barry Saltzman, Finite Amplitude free convection as an initial value problem, *Journal of the Atmospheric Sciences*, 19, (329-341) 1962.

[10] John Guckenheimer and Philip J. Holmes, *Nonlinear Oscillations, Dynamical Systems and Bifurcations of Vector Fields*, Springer-Verlag Applied Mathematical Sciences Series 42, 1983.

[11] A. C. Hindmarsch, LSODA and LSODI, two new initial value ordinary differential equation solvers. *ACM-SIGNUM Newsl. 15, (10-11), 1980.*

Monitoring Program Performance on Large Parallel Systems†

Kenneth Bobey

Myrias Research Corporation
#900 10611-98 Avenue
Edmonton, Alberta, Canada T5K 2P7

ABSTRACT

This paper describes the performance monitoring tools available for the Myrias Parallel Supercomputer. These UNIX tools allow the behavior of large parallel applications to be examined. Factors such as serial sections of code, paging patterns, task distribution, and parallel system overhead can be studied to ensure a program achieves maximum speed-up and fully exploits the parallelism provided by the architecture.

Introduction

The Myrias Parallel Supercomputer is a parallel processing architecture that is scalable to 1024 processors and beyond. The design of the entire system has emphasized ease of use. Loop constructs in Fortran 77 and C programs are replaced by *pardo* statements, which create sets of tasks that run in independent address spaces [ThM88].

Each task begins execution with a virtual copy of its parent's address space. There is no communication among sibling tasks; each executes independently of the others. At task completion the parent's memory is updated using a set of merging rules and the parent then resumes. A detailed description of the Myrias memory model can be found in an earlier paper [KVW87].

This paper describes the UNIX-based monitoring tools available for the Myrias system. First, the traditional UNIX tools for performance analysis are reviewed and it is shown why extensions are necessary for multiprocessor environments. Next, extensions that have been implemented for several parallel processing systems are discussed. The Myrias performance monitoring tools are then presented and their use illustrated with an example.

Traditional Tools

Profiling is the primary method used to identify performance problems in UNIX applications. The standard profiler, *prof* [Com86], displays profile data for programs compiled with the *-p* option. A more general profiler is *gprof* [GKM83]

† Paper prepared for presentation at the Workshop on UNIX and Supercomputers, Pittsburgh, September 1988. Limited distribution, not for public release.

UNIX is a registered trademark of AT&T Bell Laboratories. Sun Workstation is a registered trademark of Sun Microsystems, Incorporated. CONCENTRIX is a registered trademark of Alliant Computer Systems Corporation. DYNIX is a registered trademark of Sequent Computer Systems, Incorporated.

which produces extensive profiles at the function level. Execution profiles generated by *gprof* contain execution counts (the number of calls to each function), execution times, (an estimate of the amount of time spent executing each function), and call graphs, (the calling relation among functions).

A portion of the *gprof* output from a numerical program is shown in figure 1. This flat profile indicates the function *integrate* accounts for the vast majority of the execution time spent by the program. Optimizing *integrate* would be the first place to start in order to improve performance.

granularity: each sample hit covers 4 byte(s) for 0.02% of 91.48 seconds

% time	cumulative seconds	self seconds	self calls	total ms/call	ms/call	name
90.1	82.38	82.38	64	1287.19	1287.19	_integrate
9.9	91.42	9.04	1	9040.00	9040.00	_profil
0.0	91.44	0.02	19	1.05	1.05	___doprnt
0.0	91.46	0.02	1	20.00	20.00	_read

Figure 1. Sample *gprof* Output

Both *prof* and *gprof* presume an iterative approach to finding and fixing "hot spots" in user applications. Unfortunately, there are several limitations to this approach [GKM83, ArG88]:

(1) The overhead introduced by the tools can be quite high. For *gprof*, execution overhead is typically 5-30%.

(2) The intrusive nature of the tools may skew the profiled data.

(3) Recompilation is necessary to prepare the program for profiling. Once this has been done, profiling data is always generated and cannot be disabled.

(4) Profilers work best with highly modular programs and are not well suited to recursive, optimized, or heavily in-lined programs.

(5) Once a program "hot spot" has been identified, it still remains up to the user to find and implement a solution.

(6) Output can be quite voluminous, particularly in the case of *gprof*.

A descendent of the standard profilers is *mprof* [ZoH88], a tool for studying the dynamic memory allocation of a program. Instead of profiling execution time, *mprof* addresses a different set of problems but otherwise follows the same philosophy and has the same interface as *gprof*. Output produced by *mprof* consists of a memory leak table (the leaked memory allocated by each function), allocation tables, (the objects allocated by each function), and allocation call graphs, (the indirect callers of functions allocating memory).

Many of the limitations of the standard profilers are also exhibited by *mprof*. Overhead is particularly high. Typical slowdown of the application is 2-4 times but can be much greater. In addition, up to 33% can be added to the memory allocated by the application [ZoH88].

Multiprocessor Tools

UNIX has recently been ported to a variety of parallel architectures [RuW87]. There is no reason why the traditional tools cannot be used to investigate performance on these systems. For example, Sequent recommends the following multiple step procedure to select a multitasking method suitable for the DYNIX operating system [Seq86]: (i) create a program profile with *gprof*, (ii) identify subroutines that consume large amounts of execution time, and (iii) examine the call graph to determine whether these subroutines are part of independent loops that can be parallelized.

However, for parallel systems in general, there are a variety of important factors that *cannot* be measured using the existing tools.

(1) The amount of serial work in a program. The serial portion is known to be the limiting factor in how much speedup can be achieved by an application [Gus88].

(2) The program's locality of reference. As for uniprocessor systems, improving the locality will decrease paging which often has a positive effect on the performance of an application [DPS87].

(3) The distribution of tasks in the system. The task distribution should yield a uniform workload across all processors.

(4) The additional parallel processing system overheads. Of particular importance for some architectures is the synchronization overhead required to manage access to shared resources [DaH88].

A variety of new tools are available that address these issues.

IBM's Experimental Parallel Environment (VM/EPEX) contains a parallel simulation tool (PSIMUL) [DPS87] and a speedup analyzer (SPAN) [BDN87] which allow many of the above factors to be investigated in an ideal, simulated environment.

A portable tracing facility has been implemented at Stanford University that allows synchronization mechanisms to be studied on shared multiprocessor systems [DaH88]. The package has been used to investigate how locking, task size, and unbalanced computation affect the speedup of applications modeled as collections of UNIX processes.

The Parasight programming environment provides a flexible platform for building profilers for shared memory multiprocessors [ArG88]. Parasight was designed to overcome many of the limitations of the conventional tools, especially the problems of overhead, intrusiveness, and inability to selectively profile. To avoid intrusion, parasite observer programs are run concurrently with the application being monitored. Parasight has been used to implement a *gprof* style functional profiler as well as to design custom and interactive profilers.

Several vendors provide system oriented performance tools that permit the user to observe how their application utilizes critical system resources. The program can then be adjusted if resource bottlenecks are detected. The CONCENTRIX *mon* [All87] command allows system statistics to be monitoring in one of three modes: (i) for a process and its descendents, (ii) for all processes in the

system, or (iii) for all installed processors. Sequent provides the DYNIX Monit tool which examines post-mortem trace files and displays graphs of system queues and resource usage [KeS87].

Myrias Tools

The Myrias Parallel Supercomputer is designed to be both *programmable* and *scalable*. Many potential performance concerns of other architectures are not issues with the Myrias system. For example, the exclusion and synchronization properties of the Myrias memory model are handled automatically by the system. This eliminates the need for special tools to analyze such low level software and hardware primitives.

Performance tools are needed for the Myrias system for a different reason. Often programs being ported to parallel architectures have been optimized for specific serial or vector machines. To achieve optimal performance in the new environment, it is necessary to identify and fully exploit the parallelism inherent in the application. High level performance tools are required to assist the user in doing so.

Myrias offers two methods to help the user understand the degree of parallelism in an application. First, a workbench is provided to allow parallel programs to be run on a host uniprocessor system. Second, key performance statistics may be collected when programs are executed on the Myrias system.

The Myrias Parallel Programmer's Workbench (PPWB) implements the Myrias user programming model on a Sun Workstation [Myr88]. PPWB allows one to experiment with an application before actually running it on the Myrias system; it is an ideal tool for learning parallel concepts, exploring parallel algorithms, and evaluating parallel applications.

PPWB includes the Myrias Parallel C and Parallel Fortran compilers, associated libraries, and the Myrias execution supervisor *prun*. Two classes of statistics are provided by *prun*.

(1) Execution time statistics: total user time t_{total} (the sum of execution times for all parallel tasks), best execution time t_{best} (the minimum execution time that could be achieved), and potential speedup s_{pot} (the maximum speedup that could be achieved). The potential speedup is given by $s_{pot} = \frac{t_{total}}{t_{best}}$.

(2) Paging statistics: the total number of pages merged p_{merge} and the pages merged per second of user execution time p_{rate}. The paging rate is given by $p_{rate} = \frac{p_{merge}}{t_{total}}$.

Figure 2 shows *prun* statistics for the numerical program profiled previously.

total user time 105.300018
best user time 1.719997
potential speedup 61.221050
significant merges 65
significant merges per user cpu second 0.617284

Figure 2. Sample *prun* Statistics.

One may experiment with the simulator or proceed directly to running a program on the Myrias system.

It is very important to understand the performance of applications on parallel systems to ensure parallelism is fully exploited. To gain this understanding, the *mrun* command, which runs a program on the Myrias system, allows a set of orthogonal statistics to be displayed either in real time as the program executes, or in summary form upon program termination.

Statistics are grouped into classes and computed across all processing elements (PEs) on which an application is executed.

(1) Execution time statistics.

- *User CPU* - the total amount of time spent executing user code.
- *System CPU* - the total amount of time spent executing system code.
- *Idle CPU* - the total amount of time the CPU was idle.

The execution time statistics show how effectively the program utilizes available PEs. For example, high idle times may indicate large sections of serial code. Eliminating serial code will reduce idle time and possibly increase user time.

(2) Paging statistics.

- *Page ins* - the number of pages transferred into PEs.
- *Page outs* - the number of pages transferred out of PEs.
- *Pages written* - the number of pages written by user tasks.
- *Pages merged* - the number of pages formed by merging for user tasks.

The page in and page out statistics describe a program's paging activity; low page in and page out rates indicate a high locality of reference. Pages written and merged provide information about how the application uses the Myrias memory model.

(3) Task statistics.

- *Running tasks* - the number of tasks currently running.
- *Completed tasks* - the number of tasks that have been completed.

Task statistics can be used to verify the user programming model and are especially useful for monitoring the progress and behavior of an application in real time.

(4) I/O statistics.

- *Reads* - the number of read requests.
- *Bytes read* - the total number of bytes read.
- *Writes* - the number of write requests.
- *Bytes written* - the total number of bytes written.

These statistics gauge the program's I/O rate. The average number of bytes per request indicate whether I/O is being performed efficiently. Programs with heavy I/O requirements can increase the number of bytes per request thus improving the efficiency.

When more detailed monitoring is needed, *mrun* command line modifiers may be used to display the average, variance, maximum, and minimum value of each statistic.

The numerical program illustrates the usefulness of the *mrun* statistics. The first time this program was run on the Myrias system an unusually high idle time was noted as shown in figure 3.

08:34:45 - user 6.74s (48.2%) system 2.44s (17.4%) idle 4.82s (34.4%)

Figure 3. Sample *mrun* Statistics (I).

Examining the task statistics in real time revealed the presence of "straggler" tasks due to an uneven distribution of work across PEs. As figure 4 illustrates, modifying the algorithm to utilize tasks of approximately the same size not only reduced the idle time but also decreased the total execution time for the program considerably.

08:33:59 - user 6.25 (73.6%) system 1.05 (12.3%) idle 1.20s (14.1%)

Figure 4. Sample *mrun* Statistics (II).

Summary

This paper has presented the performance monitoring tools for the Myrias Parallel Supercomputer. The tools overcome the limitations of the traditional UNIX profilers and permit the measurement of several important performance factors in a parallel environment. In addition, the Myrias tools assist the user in exploiting parallelism and achieving maximum speedup for an application.

Acknowledgements

Thanks are due to the many members of the Software Department at Myrias who contributed to the concepts and design presented here.

References

[All87] Alliant Computer Systems, *CONCENTRIX Commands and Applications Manual*, Alliant Computer Systems Corp., Littleton MA, October 1987. Version 3.0.

[ArG88] Z. Aral and I. Gertner, Non-Intrusive and Interactive Profiling in Parasight, *SIGPLAN Notices: Proceedings of the ACM PPEALS 1988 23*, 9 (September 1988), pp. 21-30.

[BDN87] A. Bolmarcich, F. Darema, V. Norton and K. So, A Speedup Analyzer for Parallel Programs, in *Proceedings of the 1987 International Conference on Parallel Processing*, S. Sahni (ed.), Pennsylvania State University Press, University Park PA, August 1987, pp. 653-662.

[Com86] Computer Systems Research Group, *UNIX User's Reference Manual*, University of California, Berkeley CA, April 1986. 4.3 Berkeley Software Distribution.

[DPS87] F. Darema-Rogers, G. Pfister and K. So, Memory Access Patterns of Parallel Scientific Programs, *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems 15*, 1 (May 1987), pp. 46-58.

[DaH88] H. Davis and J. Hennessy, Characterizing the Synchronization Behavior of Parallel Programs, *SIGPLAN Notices: Proceedings of the ACM PPEALS 1988 23*, 9 (September 1988), pp. 198-211.

[GKM83] S. Graham, P. Kessler and K. McKusick, An Execution Profiler for Modular Programs, *Software - Practice and Experience 13*, 8 (August 1983), pp. 671-685.

[Gus88] J. Gustafson, The Scaled-Sized Model: A Revision of Amdahl's Law, *Proceedings of the Third International Conference on Supercomputing II*, (1988), pp. 130-133.

[KeS87] T. Kerola and H. Schwetman, Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs, *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems 15*, 1 (May 1987), pp. 163-174.

[KVW87] A. Kobos, R. VanKooten and M. Walker, The Myrias parallel computer system, in *Algorithms and Applications on Vector and Parallel Computers*, H. Riele, T. Dekker and H. Vorst (ed.), North-Holland, Amsterdam, 1987, pp. 103-118.

[Myr88] Myrias Research Corporation, *Parallel Programmer's Workbench Guide*, Myrias Research Corp., Edmonton AB, 1988. 5-14200-M2 Preliminary.

[RuW87] C. Russell and P. Waterman, Variations on UNIX for Parallel-Processing Computers, *Communications of the ACM 30*, 12 (December 1987), pp. 1048-1055.

[Seq86] Sequent Computer Systems, *Guide to Parallel Programming*, Sequent Computer Systems, Inc., Beaverton OR, 1986.

[ThM88] C. Thomson and K. McPhee, Experience Scaling the Myrias System to Hundreds of Processors, *Proceedings of the Third International Conference on Supercomputing II*, (1988), pp. 125-129.

[ZoH88] B. Zorn and P. Hilfinger, A Memory Allocation Profiler for C and Lisp Programs, in *Proceedings of the Summer 1988 USENIX Conference*, USENIX Association, Berkeley CA, June 1988, pp. 223-237.

Some Observations on Computer Performance Characterization: Supercomputer and Mini-supercomputer Clocks and Compilers

E. N. Miya

Computational Research Branch
NASA Ames Research Center
Moffett Field, CA 94035
eugene@ames.arpa
UUCP: ames!eugene

ABSTRACT

There are many reasons for running benchmarks on computers. Unless care is taken throughout the entire performance measurement operation, the results may not provide the understanding sought.

This small empirical study concentrates on distinguishing "between" computer systems. More precisely, the syntax of system timing and how that syntax can also reveal information about compiler quality is considered. A simple test program reveals significant system artifacts. Replacing statistics with graphics provides insight about the different architectures. These observations have implications on conventional empirical measurement technique and on future measurement issues.

Introduction

There are many reasons for running benchmarks on computers. Common reasons include selection, tuning, and verification of systems. Unless care is taken throughout the *entire* performance measurement operation, the results may not provide the understanding sought. Misunderstandings result because of oversimplified tests, portability problems, and running application programs tuned to specific hardware and software.

The *machine* measurement problem must separate from the *application* problem. In this paper, the application is ignored, so that benchmarking can be systematic. Parts "between" computer systems are then analyzed. Performance measurement must have at least, an awareness of *all* the parts. If components are placed into computer hardware or software to improve speed, like a cache or compiler, then these features might have measurable performance *artifacts*. Then, the reassembly of the pieces will allow improved study of the applications.

Some benchmarking and calibration problems are surveyed first. The syntax of system timing and how that syntax can also reveal information about compiler quality is considered. A simple test program revealed significant system artifacts through measurement. Seven machines and several compilers were tested showing major qualitative differences in the hardware and software. These differences will appear without statistical averaging.

Benchmarking

Many programs are designed to measure performance. Every program is potentially a benchmark, but synthetic benchmarks abound (e.g., the Whetstone [Curnow], the Gibson mix [Gibson], the Livermore loops [McMahon], the Dhrystone [Weicker], the Linpack [Dongarra], the NAS kernels [Barton]) as well as full-scale applications [Jordan]. These programs have a variety of problems including portability, influence, interpretation, and content among others.

The first problem is portability, and small size is the standard solution. Benchmarks are usually small to minimize changes for transport to and execution on many computers. Portability also implies a minimum of input data. Benchmarks, like all programs, need data which is either self-contained in source code, self-generated, or read in by I/O. This data eventually influences program execution, and the next problem: influence.

The next problem as long-time benchmarkers know, is that compilers and operating systems influence measurement consistency. Benchmarks are subject to many other influences, e.g., hardware features, but few benchmarks specifically test or react to these features. Some influences induce penalties or overheads whereas others are beneficial, e.g., compiler optimization, as will be shown. These influences require control to achieve consistent "apples to apples" comparison conditions.

Another problem is that benchmarks are subject to misinterpretation for reasons of content and oversimplification. For instance, many benchmarks such as Linpack [Dongarra] or the Sieve of Eratosthenes return a *single figure of merit* . While a single number is easy to read, it cannot reveal more than simple ordering. Statistical analysis is frequently used, but this creates other problems, e.g., what type of (statistical) mean to use [Worlton],[Wallace]. There must be a better way to obtain content.

Content means: what requires measuring? What is measured? The benchmarks surveyed all claim to measure "CPU and memory" performance. But, do they really? There is little sound analysis for these claims. A single figure statistic is generated, but little insight is obtained. Consider, there is a bias for CPU/memory benchmarks, but notably, few I/O benchmarks exist. So, if application programs require characterization, these issues need blending.

The question is: where to start? There are too many facets to detail in a short paper, but two interesting features are system clock and compiler quality. More precisely, the syntax of system timing is considered and how that syntax can also reveal information about compiler quality. Clocks deserve examination, because other measurements are based on them. Poor clocks require test codes which are repeated many times during a single interval. The interval is then divided by the number of repetitions to obtain a measurement. This is not an average, and it possibly introduces new influences. Good time keeping balances accuracy, resolution, and reproducibility [Jespersen]. This is a form of *calibration* and a simple test program reveals significant system artifacts.

Compiler optimization is another good feature (artifact). Harbison surveyed some compiler optimizations [Harbison]. Two optimizations are "common subexpression elimination" and "inline code substitution" for subprograms. These two

optimization are easily testable. But first, a description of computer time keeping is needed.

A Short Tour Through Time

There are many ways to get time from a computer system. Timing *appears* trivial, except that there are no standards. Time on a computer is very machine dependent. It has syntactic and semantic aspects. The semantic ones include clock resolution, skew, accuracy, and synchronization. The syntactic aspects consist of language structures used to obtain time: function or subroutine calls, and type variables (integer or floating point [single or double precision]) or structures (variables, arrays, or records).

The syntax to obtain time is summarized with two questions: "How is time requested (called)?" and "How is the time returned?" Each has two forms. The first and most frequent calling form is a function or procedure call which returns a *time value*. A clock must update that value, and it should, but not always, update constantly, i.e., tick. The time value is either an *incrementing register* or a *word of memory*, the first form of return, which can be *recorded* for measurement. Unfortunately, word and register sizes vary. This imposes precision limits, but time returned using multiple array or structure elements, the second form of return, can solve this problem. The Alliant FX/8™ [Perron] returns a two-element array giving a 10-microsecond increment.

A notable clock is found in Berkeley Unix™. The timing call *gettimeofday*(2) is a simple procedure call, but the operating system attempts to maintain time in a machine-independent form. This form is represented by a multi-word C structure (a record) in a file named "time.h" containing:

```
struct timeval {
        long  tv_sec;           /* seconds */
        long  tv_usec;          /* and microseconds */ };
```

Note that the use of a microsecond field does not imply that a machine has a microsecond clock. The gettimeofday(2) clock is a software clock, and its name is merely a format for translating memory. It is a field approximating a measure formatted to microseconds. Hence, 100 Hz clocks increment by 10000s. This is a compromise for hardware clocks. Future supercomputers will require this structure in hardware, and it will require the addition of a picosecond field to the structure.

A macro processor can hide differences in clock syntax by making text substitutions prior to any compilation. The M4 macro processor can prepare most test programs by using a workstation prior to export. Macro processors are recommended for this purpose, since they can also minimize overheads and still hide detail.

Another form is a LISP-style "scoped" call which returns a *timed interval*. For instance, if *command* is the object of measurement, then *time command* is syntax of measurement. This form is seen on LISP and Unix systems. This differs from the first syntactic form which returns a time value or date. The difference between two points in time constitute an *interval* which brings up the importance of the semantics of time.

An interval is assumed to have a test code running between two points of time [Figure 1]. This has an assumption of synchrony: all work is completed when the stopping time is recorded. This assumption is potentially nullified in asynchronous, data-driven systems [Treleaven]. Synchronization is important when measuring concurrent or parallel programs, and to insure consistent pre-test conditions. Synchronization should be the last thing done before a measurement. This must be assured in asynchronous systems.

Malony introduces and defines the semantic concept of *timing scope* which will gain importance in asynchronous systems, since users must insure that work is complete [Malony]. The semantic difference results from the distinction between measuring *interval* and the technical term "*date*" (a point of time from some reference) [Jespersen]. The semantics of time are considerably more complex than mere scope.

The closest syntactic proximity of clock are two consecutive time calls results in "resolution:"

```
get_time(time1)
get_time(time2)
```

This is the minimum syntactic resolution of a clock. Time returned as the result of a function call has one distinct advantage over explicit call clocks: values don't require explicit assignment of storage, keeping grain small. This is important in synchronization when using tight spin-wait loops, for example:

```
      call time(time1)
10    call time(time2)
      if(time1 .eq. time2) go to 10
```

is not as syntactically tight as:

```
      time1 = get_time()
10    if(time1 .eq. get_time()) go to 10
```

Now that the timing assumptions are understood, the subject of test content is raised.

Measurement Code

There are many ways of checking clock quality, and this test times subprogram call overhead. The method is to induce artificial work to see if the time scales with work, or if a compiler is smart enough to eliminate this work. First, the minimum resolution is obtained for two successive clock calls without intervening subprograms or procedures. This corresponds to depth 0 (zero) call:

```
get_time(start)
get_time(intermediate)
get_time(stop)
...
get_time(start)
call one
get_time(stop)
...
```

```
get_time(start)
call two
get_time(stop)          # recording/output code removed for
...                     # readabilit
get_time(start)
call three
get_time(stop)
...
```

The next section of code calls a parameterless subprogram: (declarations and output are removed for clarity)

```
subroutine one
common intermediate
get_time(intermediate)
return
end

subroutine two
call one
return
end

subroutine three
call two
return
end

subroutine four
      ...
end

subroutine five
      ...
subroutine six
...
```

The entry into and exit out of subroutines is timed. It is reasonable to assume that the time in equals the time to get out, but results show that is not always the case. Clock synchronization occurs prior to the start of every measurement to insure consistent pre-test condition. This program is very simple, yet reveals some interesting things about clocks and compiler technology. Lastly, the entire code is repeated three times for different samples.

If execution time is plotted against the call depth, a *characteristic curve* results. These are not truly curves, since the depth of a subroutine call is not continuous, but the space filling line improves visibility. Curves have two notable attributes: first is measurement "noise" and the second is the curve "slope" or "flatness." Noise, deviation, or variation represent *clock skew* between samples and define the resolution and activity of the measurement environment. Deviations may appear consistent or at random. An observer should see constant sample intervals of linearly increasing cost with call depth. "Noise" should not be present in a reproducible environment.

The slope attribute tests compilers. Older generation compilers should take longer to execute deeper subroutine calls. Execution should plot a linear time increase [Figure 2, below]. Newer "smart" compiler technology can reduce subprogram overhead time by optimizing with inline code substitution or constant subexpression elimination. If subprograms are eliminated, there should be no time cost, hence, the graph of execution time should be flat [Figure 2].

There are several other influences. First, the order of subprogram compilation can have significance (i.e., compile order: one through six or six through one). This is because clever compilers and forward defining languages (like Pascal and Ada™) make optimization easier. This makes testing harder. Several methods can test if this happens: 1) symmetrically reverse the sequence of source code or 2) resort to separate compilation and linking. However, the latter is not a perfect test as some systems use separable environmental information allowing the passing of information to aid optimization. No systems in this study exhibited the need to check this effect. The last influence can appear from printing output, time intervals. So output is separated from other executable statements to minimize I/O system interference.

Observations

Most of these observations represent "between-machine" comparisons. There are also significant "within-machine" comparisons. There is no overlay of different characteristic signatures, since there are different environmental characteristics. The machines tested include the Cray X-MP™, the CDC Cyber 205™, the Convex C-1™, the IBM 3090™, the Alliant FX/8, the Cydra 5™, and the Amdahl 1200™.

Cray X-MP

The Cray X-MP is a classic supercomputer. It is a short-vector multiprocessor which comes with either an 8.5 or a 9.5 nanosecond instruction cycle. Most instructions are executed in a single cycle [Larson]. The X-MP is an especially good architecture to study performance measurement, because it has a minimally intrusive performance measurement monitor [Larson86].

The X-MP has two clock interfaces. The *real-time clock*.(RTC) is a precision hardware clock. It is fast: accessible in a single CPU cycle. The *SECOND* clock is another interface which counts system overhead. The RTC function calls do not use OS overhead and simply gets the wall clock time. They can be checked against one another simply by timing each other:

```
START = SECOND()                ISTART = IRTC()
IDUMMY = IRTC()                 DUMMY = SECOND()
STOP = SECOND()                 ISTOP 1_ IRTC()
RMEAS = STOP - START            MEAS = ISTOP - ISTART
```

The left measurement yields 0.0 seconds, but the right measurement yields 1680 cycles.

Clock Quality (Overlap): The first notable characteristic is that three samples have identical signatures [Figures 3 and 4]. They are not averaged: each line is actually three perfectly overlapped lines. This is regardless of clock type (RTC or

SECOND), or whether entry or exits are timed. This show good reproducibility and is a good clock environment.

The second characteristic is that the curve is not precisely linear. There are slight, but consistent perturbations from predicted straight lines of Figure 2. The reason for this is not known.

The third characteristic is that the entry time interval is not the same as the exit interval. This is also dependent on clock type (RTC or SECOND, a "within-subject" characterization). Two different clock calls produce conflicting interpretations: the RTC implies that exiting routines takes slightly longer than entering whereas the SECOND clock has the inverse case. Also note: the RTC resolution has near zero access time for no subroutine call, time axis intercept, whereas the SECOND clock has a measurable resolution of about 0.27×10^{-5} seconds.

Compiler Quality (Slope): A steadily increasing line shows the CFT compiler is an older technology and does not optimize subroutine overheads. Operating systems for the X-MP do not page (virtual memory), and consequently, there are few operating system influences.

CDC Cyber 205

The Cyber 205 is a long-vector uniprocessor supercomputer [Lincoln]. It has a paging operating system (VSOS™) which might be a source of clock skew.

Clock quality: The Cyber 205 has a 20 nanosecond machine cycle with a microsecond resolution clock. A cycle-time clock is not present on the machine. Sample fluctuation (noise) is visible [Figure 5], so some clock skew is present.

Compiler Quality (Slope): The entry time versus exit time are roughly the same unlike the X-MP, so only one set of samples is presented (the exit overhead time). The compiler does not appear to optimize the code, hence, it uses an older compiler technology.

Convex C-1

The Convex C-1 [Wallach] is one of the first of the new class of *mini-supercomputers* or "*Crayettes*." It runs a Berkeley UNIX kernel, and it pages. The Convex has the previously mentioned Berkeley time.h structure which will become more prevalent.

Compiler Quality (Slope): The Fortran compiler is one of the new generation, advanced compilers. It demonstrates this technology optimizing the call overhead away [Figure 6]. The compiler appears to keep the first call (initial slope to a "knee" of performance).

The interesting progression is the increasing execution time with subsequent samples [1 through 3]. This performance is unexpected. Company representatives reported that this was a cache hardware problem which was subsequently fixed. The new curves all correspond to the "Sample 1" curve. This test illustrates the need for diagnostic benchmarking.

Clock Quality: The Convex-1 has a micro-second clock with a 100 nanosecond machine cycle. There is little random noise in this sample implying a good measurement environment. The quality of the clock is good with minimal skew.

The Convex C-1 gave the opportunity for another within-machine comparison using a second language (C) with an advanced compiler [Figures 7 and 8]. The cache problem was repaired and has a characteristic curve similar to the Fortran compiler [Figure 6], but an interesting artifact is the depth 4 call which is even more expensive than a 6 deep call. The clock skew does not appear as good in the C environment, the machine appears noisier.

IBM 3090

The IBM 3090 is a high-performance mainframe computer system. It has an optional vector processing facility. It has a paging operating system. It does not have special monitoring hardware.

Clock quality: The Model 200 has an 18.5 nanosecond machine cycle with a microsecond resolution clock [Figure 9]. Clock resolution is good, but not consistent like the X-MP.

Compiler Quality (Slope): IBM vector FORTRAN has an advanced compiler. The curves are all flat, without the single call cost associated with the Convex C-1.

Alliant FX/8

The Alliant FX/8 [Perron] is another "first-generation minisupercomputer". It is a vector multiprocessor. The architecture involves heterogeneous processors (Interactive Processors [IPs] and Computational Elements [CEs]).

Clock quality: The clock has a 10 microsecond resolution [Figure 10]. There appears to be little skew. This is a somewhat coarse clock resolution but sufficient for purposes of this study.

Compiler Quality: The compiler optimizes the test code (i.e., the curve is flat), but it appears to be have a consistent initial cost for calling subroutines with subsequently faster calls. This is indicative of cache and virtual memory systems. These systems can require cache or a page start-up costs.

This type of program execution is ill-suited for benchmarks which repeat tests to get a measurement. The first execution might be more typical of a pass through a loop than subsequent repetitions which mis-sample measurement sections.

Cydra 5

Cydra 5 is a "second-generation minisupercomputer" [Rau]. It has a characteristic similar to the Alliant [Figures 11 and 12]. It is regarded by its developers as a coarse-grained data flow machines.

Clock Quality: Generally, good. Reproducible timing with little clock skew. One data point might represent some system noise.

Compiler Quality: Good. It appears comparable to the other minisupercomputers.

Amdahl 1200/Fujitsu VP-200

The Amdahl 1200/Fujitsu VP-200 is a computer based on vector processing [Miura]. It is a long-vector uniprocessor regarded as a "super" rather than a minisupercomputer.

Compiler Quality: Good. The Japanese compilers on the VP-200/1200 appear [Figures 13 and 14] to optimize code.

Clock Quality: The VP clock does not have the running consistency of the X-MP clock. The timing environment appears similar to the Cyber 205.

Discussion

This short study shows that careful measurement technique is needed. Computer systems have widely varying characteristics. This work is limited, but it illustrates the problem with "single figure of merit" measurements and the limitations of statistical technique. Supplemental tools are needed, not the discreditation of statistics.

Replacing statistics with graphics provides insight about different architectures. Characteristic curves are like "electro-cardiograms" for their respective computers. Human pattern matching skills have advantages conveying information. This is similar to cardiologists' viewing cardiograms to understand heart performance. Statistics like the variance have a difficulty noting simple trends as in the Convex, Alliant, and Cydra cases. Variance can only provide limited measurement about the spread of samples and their noise, but cannot easily show trends. Users must learn the value of measurement samples not just statistics.

Lastly, the users who do benchmarking can learn from those working on functional testing. There is a tendency to assume that system software such as compilers and operating systems is optimally working when first delivered. There is no standard set of tests to validate hardware performance. Standard language validation suites exist for checking the conformance of compilers, checking machine dependences, etc. Many of these suites are quite extensive, and they are consistent. For performance, individual computer manufacturers are left to either develop their own set of tests or run large applications programs through their machines. Manufacturers then duplicate efforts and introduce performance test inconsistencies. Similar consistent test programs or suites for optimization and hardware testing will also prove valuable. It is ironic that the functional testing community has not had a greater impact on performance testing.

Conclusions

This study is a portion of a larger work-in-progress on benchmarking's problems. There is significant variation in the quality of various computer measurement environments. Measurement quality requires improvement as well as the quantity of measurement. More rigor and analysis are needed to improve measurement. Many more measurements are needed to develop better descriptions of architectures. Benchmarking's empirical foundations require improvement. These observations have implications on existing and on future measurement techniques.

Significant architectural features are visible on careful measurement. Performance is a composition of many factors which require measurement and "reassembly." The black-box is decomposable: working parts are comprehensible. Once all components are understood, the *real* problem of application benchmarking will open.

Performance measurement problems will consist of many moving targets. Computer hardware, software, and applications are all changing. Future architectures offer many interesting problems for measurement and testing. These enigmas include the many parallel architectures like hypercube and data flow processors, LISP, and graphics systems [Abu-Sufah], [Gallivan], [Malony], [Pfister], [Seitz], [Treleaven], [Gabriel], [Croll].

Acknowledgements

For time on their machines and their assistance: Rafael Saavedra-Barrera (UC Berkeley); A. Karp (IBM); R. Wilson (Zero One); R. Towle (Cydrome); R. Kolstaad and P. Smith (Convex); K. Miura, J. Roberts, C. Pangali, and M. Baldwin (Amdahl). Special thanks goes to George Michael and Ken Stevens.

References

[Harbison] Samuel P. Harbison. *A Computer Architecture for the Dynamic Optimization of High-Level Language Programs*, CMU-CS-80-143, PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, Sept. 1980.

[Jespersen] James Jespersen and Jane Fitz-Randolph. *From Sundials to Atomic Clocks: Understanding Time and Frequency*, Dover Publications, New York, 1977.

Computers in This Study

[Larson] John L. Larson. "Multitasking on the Cray X-MP-2 Multiprocessor." *Computer*, vol. 17, no. 7 (July 1984), pages 62-69.

[Larson86] John Larson. "CRAY X-MP Hardware Performance Monitor" *Cray Channels*, (Winter 1986), pages 18-19.

[Lincoln] Neil R. Lincoln. "Technology and Design Tradeoffs in the Creation of a Modern Supercomputer," *IEEE Transactions on Computers*, vol. C-31, no. 5 (May 1982), pages 349-362.

[Miura] Kenichi Miura and Keiichiro Uchida. "Facom Vector Processor System: VP-100/VP-200," NATO ASI Series, vol. F7, Springer-Verlag, Berlin, 1984.

[Perron] Robert Perron and Craig Mundie. "The Architecture of the Alliant FX/8 Computer," *IEEE COMPCON'86*, (February 1986), pages 390-393.

[Rau] B. R. Rau. "Cydra 5 Directed Data Flow Architecture," *IEEE COMPCON'88*, Feb./March 1988, pages 106-113.

[Wallach] Steve Wallach. "The Convex C-1 64-bit Supercomputer," *IEEE COMPCON'85*, (February 1985), pages 122-126.

Existing Benchmarks

[Barton] John T. Barton and David H. Bailey. "The NAS Kernel Benchmark Program," NASA TM 86711, August 1985.

[Curnow] H. J. Curnow and B. A. Wichmann. "A Synthetic Benchmark." *The Computer Journal*, vol. 19, no. 1 (February 1976), pages 43-49.

[Dongarra] Jack J. Dongarra. "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment." *ACM Computer Architecture News*, vol. 13, no. 1 (March 1985), pages 3-11.

[Gibson] Jack C. Gibson. *The Gibson Mix*, IBM TR 00.2043, Poughkeepsie, NY, June 1970.

[Jordan] Kirk Jordan. "Performance Comparison of Large-Scale Scientific Computers: Scalar Mainframes, Mainframes with Integrated Vector Facilities, and Supercomputers," *Computer*, vol. 20, no. 3 (March 1987), pages 10-23.

[McMahon] Frank H. McMahon. *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*, UCRL-53745, LLNL, Livermore, CA, December 1986.

[Weicker] Reinhold P. Weicker. "Dhrystone: A Synthetic Systems Programming Benchmark." *Communications of the ACM*, vol. 27, no. 10 (October 1984), pages 1013-1030.

Statistics in Benchmarking

[Wallace] John Wallace and Philip J. Fleming. "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results." *Communica tions of the ACM*, vol. 29, no. 3 (March 19846, pages 218-221.

[Worlton] Jack Worlton, "Bottleneckology: Evaluating Supercomputers," *IEEE Compcon Proceedings*, San Francisco, 1985, pages 405-406.

Advanced Architectural Problems for Benchmarking

[Abu-Sufah] Walid Abu-Sufah and Allen D. Malony. "Experimental Results for Vector Processing on the Alliant FX/8," TR 539, Center for Supercomputer Research and Development, University of Ill., Urbana-Champaign, IL, February 1986.

[Croll] Brian Croll, Ed Post, and the Bay Area ACM/SIGGRAPH Technical Interest Group on Performance Evaluation of Computer Graphics Systems, internal communication, croll@sun.com.

[Gabriel] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*, MIT Press, Cambridge, MA, 1985.

[Gallivan] Kyle A. Gallivan, William, Jalby, Allen D. Malony, and Pen-Chung-Yew. "Performance Analysis on the Cedar System," TR CSRD 680, Center for Supercomputer Research and Development, University of Ill., Urbana-Champaign, IL, September 1987.

[Malony] Allen D. Malony. "Cedar Performance Measurement," TR CSRD 579, Center for Supercomputer Research and Development, University of Ill., Urbana-Champaign, IL, June 1986.

[Pfister] G. F. Pfister and others. "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture." *International Conference on Parallel Processing*, IEEE, August 1985, pages 764-771.

[Seitz] Charles L. Seitz. "The Cosmic Cube." *Communications of the ACM*, vol. 28, no. 1 (January 1985), pages 22-33.

[Treleaven] Philip C. Treleaven, David R. Brownridge, and Richard P. Hopkins. "Data-Driven and Demand-Driven Computation." *Computing Surveys*, vol. 14, no. 1 (March 1982), pages 93-143.

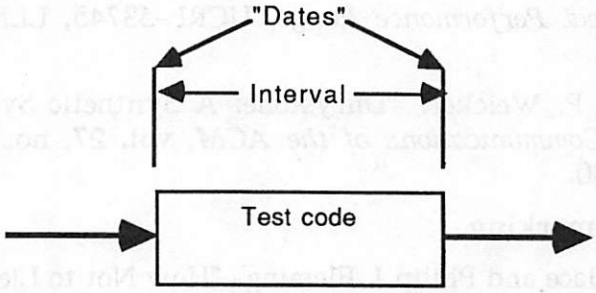


Figure 1.

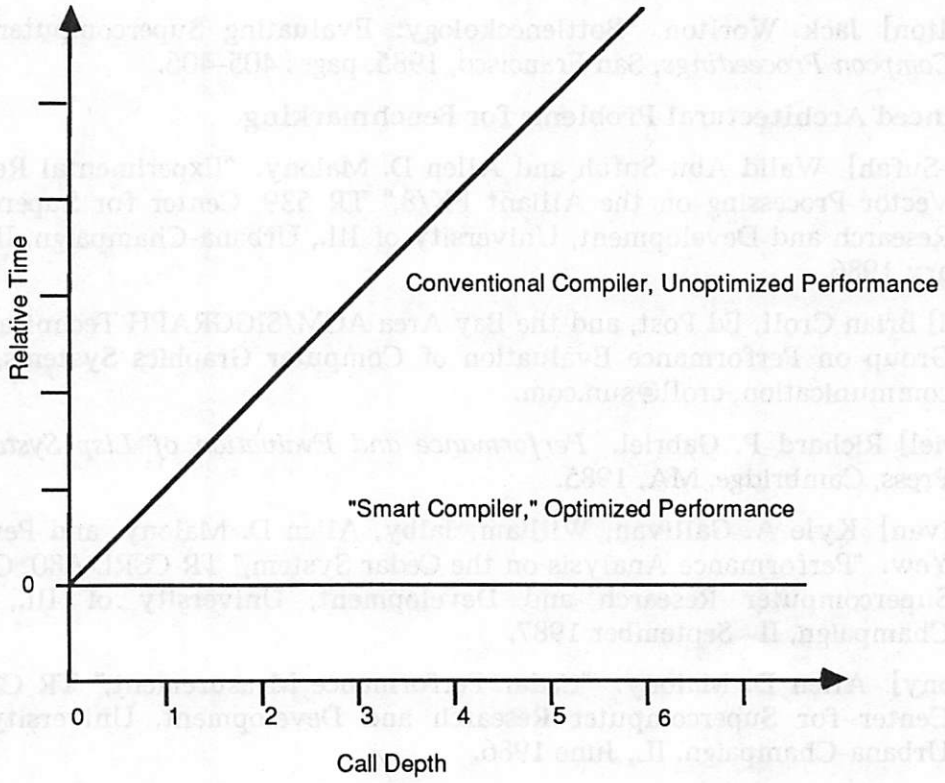


Figure 2.

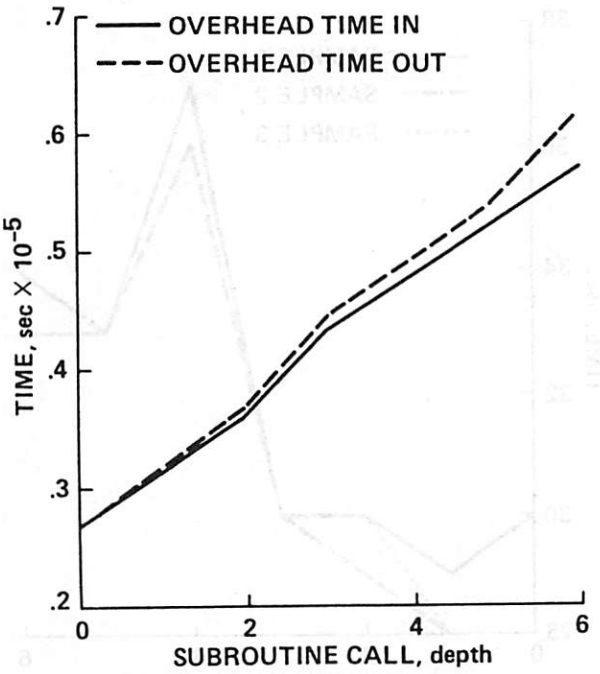


Fig. 3. CRAY X-MP [Second ( ) Clock]

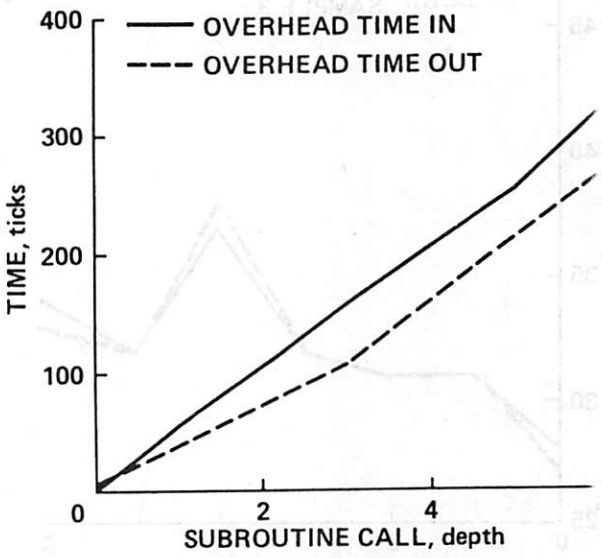


Fig. 4. CRAY X-MP [IRTC ( ) Clock]

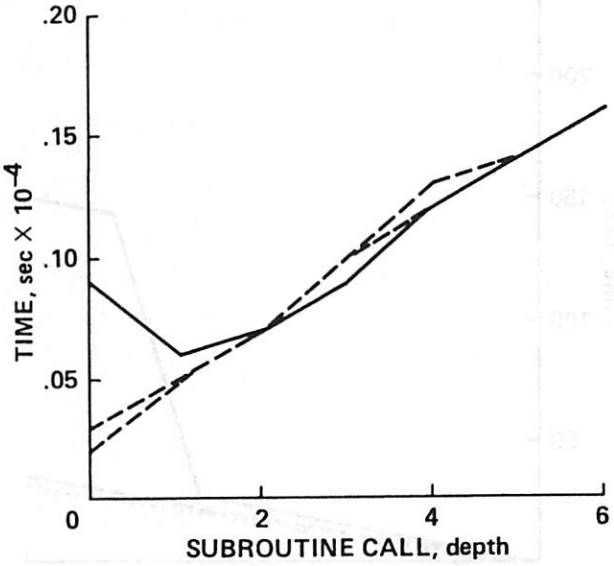


Fig. 5. CYBER 205, FORTRAN 200

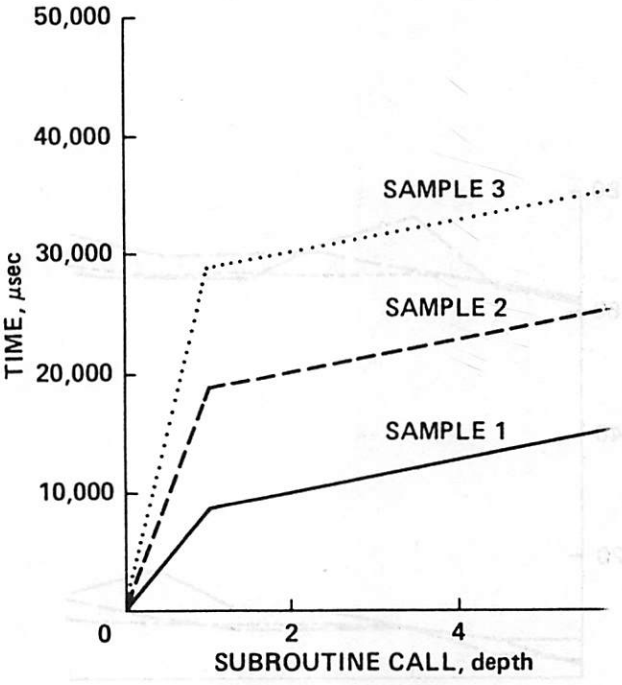


Fig. 6. CONVEX C-1, FORTRAN

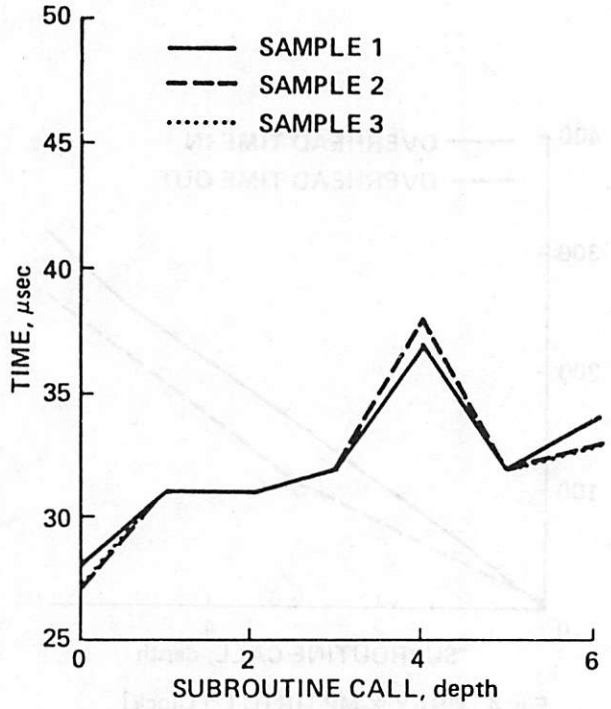


Fig. 7. CONVEX C-1, C compiler

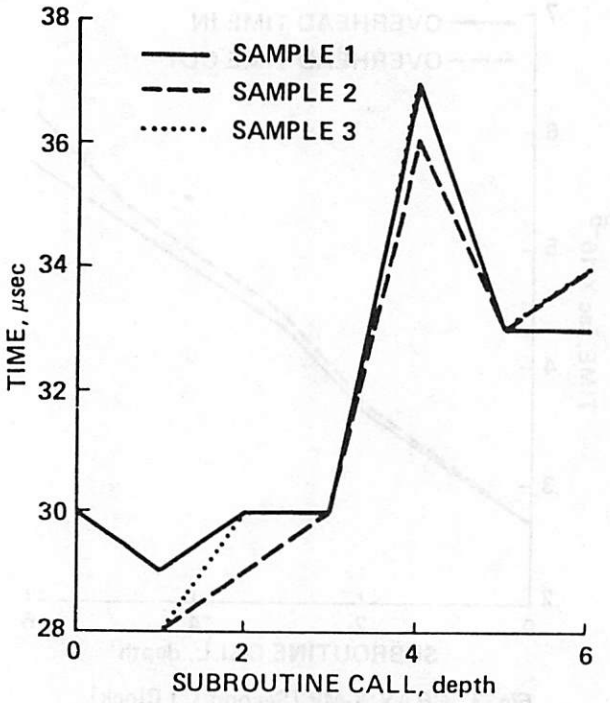


Fig. 8. CONVEX C-1, C compiler (outbound)

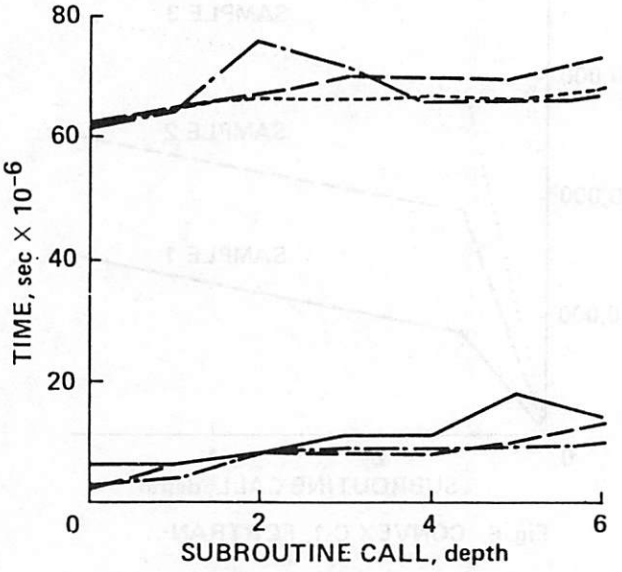


Fig. 9. IBM 3090, FORTRAN

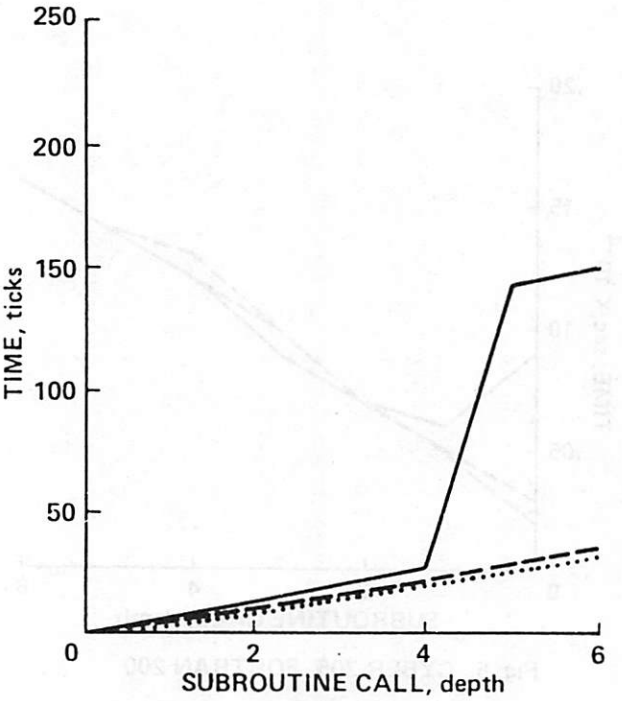


Fig. 10. ALLIANT FX/8, FORTRAN Compiler

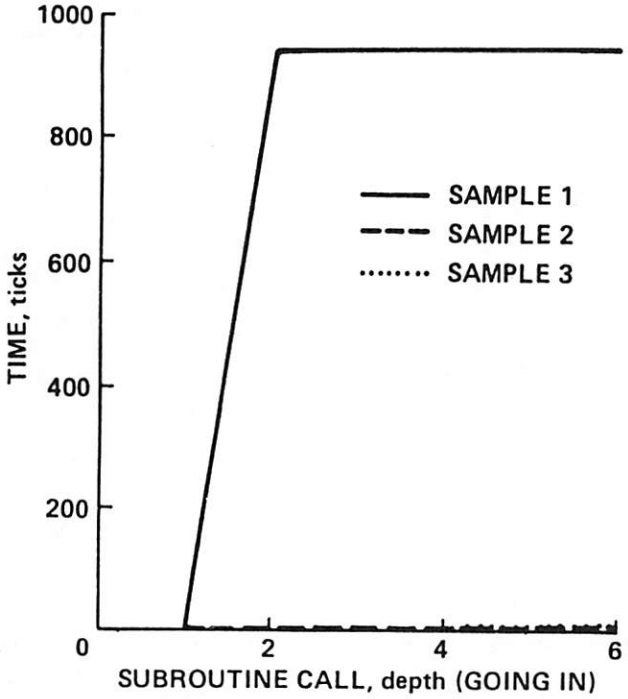


Fig. 11. CYDRA 5, Fortran Compiler

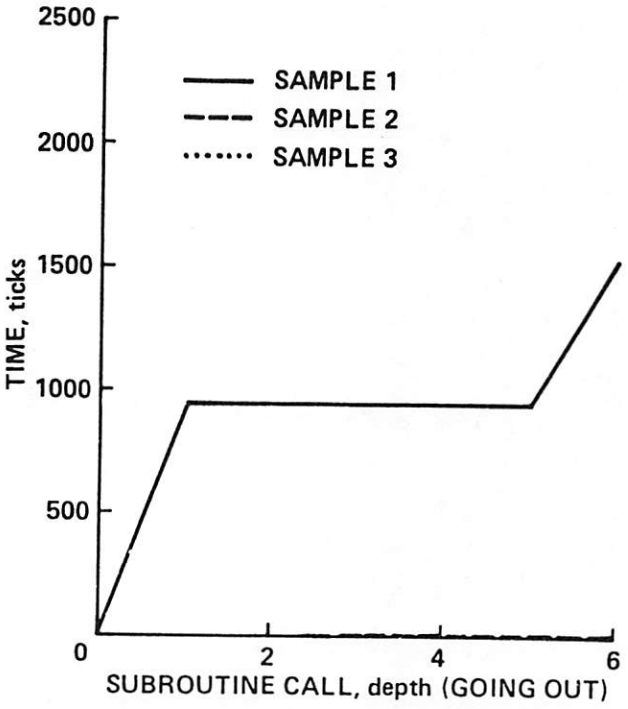


Fig. 12. CYDRA 5 (outbound)

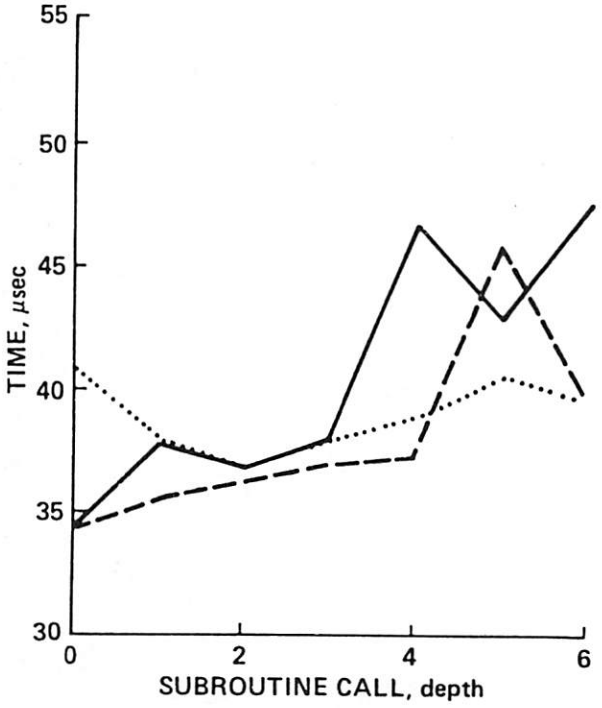


Fig. 13. Amdahl 1200/FACOM VP 200

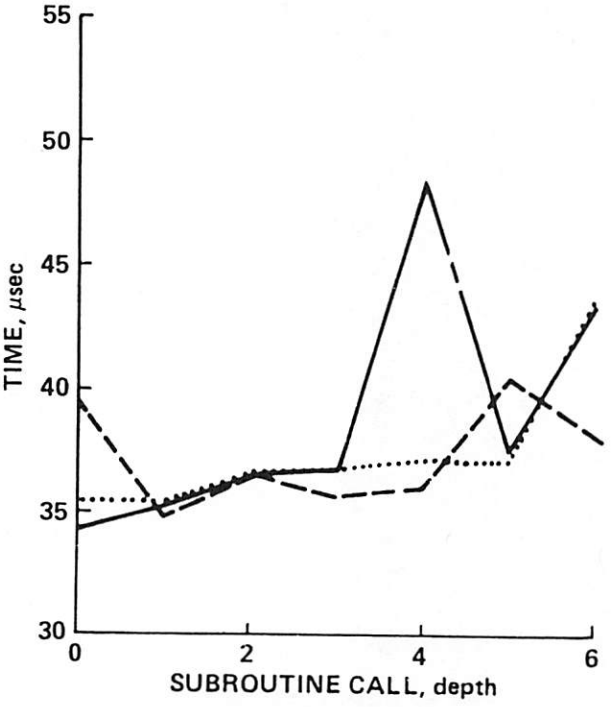
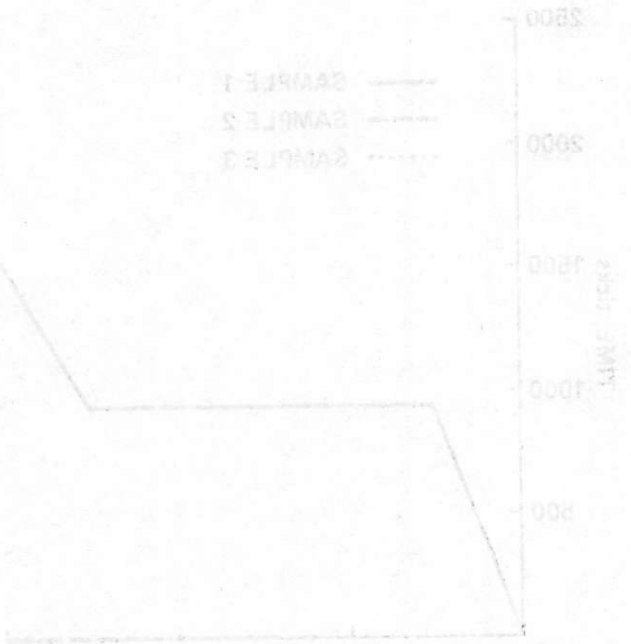
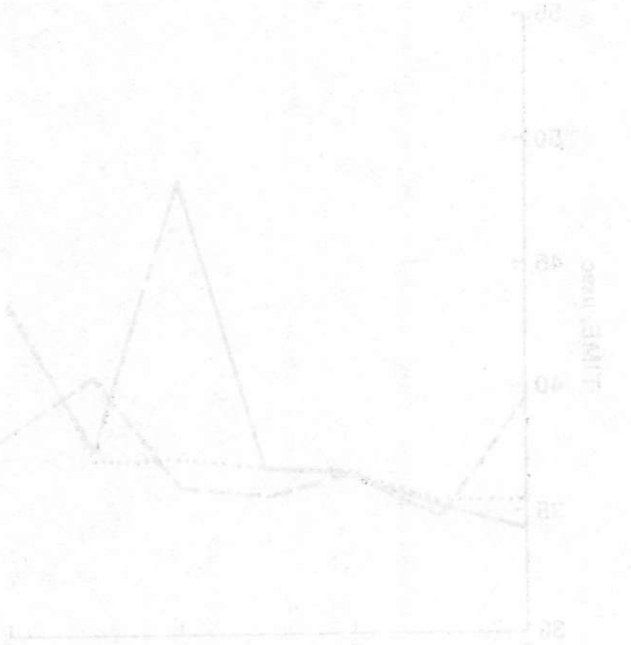


Fig. 14. Amdahl 1200/FACOM VP200





High-speed networking with Supercomputers

John Renwick

Software Divsion
Cray Research, Inc.
1440 Northland Drive
Mendota Heights, MN 55120
jkr%hall.cray.com@uc.msc.umn.edu

ABSTRACT

Supercomputers with high-speed I/O capability as well as processing speed can communicate with each other at speeds unheard-of only a few years ago. Cray Research has been working for several years to make standard network protocols run at hardware speeds. Our experience has taught us that protocols themselves are not inherently slow, but their implementations often are. Where the protocol presents limitations, there are sometimes compatible extensions that can be made to overcome them.

The RP3 Parallel Computing Environment

Ray Bryant

IBM Research Division
Thomas J. Watson Research Center, Hawthorne
P. O. Box 704
Yorktown Heights, New York 10598
raybry@ibm.com

ABSTRACT

RP3, the Research Parallel Processing Prototype, is a research vehicle under development at the IBM Thomas J. Watson Research Center for exploring the hardware and software aspects of highly parallel computation. RP3 is a shared memory machine that is designed to be scalable to 512 processors; a 64 processor machine is currently under construction. The primary method of accessing RP3 is via RT PC workstations on a local area network; RP3 is regarded as a "cycle-server" on this network. This is achieved by having RP3 and the workstations run a common version of UNIX: the Mach system from CMU. Eventually, we hope to improve system integration by using the AIX Transparent Computing Facility (a product-level implementation of Locus). The Mach/RP3 kernel is being modified to exploit features of RP3 such as local and interleaved storage, explicit processor allocation, and changes in scheduling for high-performance batch execution. This paper describes these components of the RP3 Computing Environment, the status of their implementation in RP3, and the research issues we are pursuing to evaluate these concepts.

1. Introduction

RP3, the Research Parallel Processing Prototype, is a research vehicle for exploring the hardware and software aspects of highly parallel computation. RP3 is a shared memory machine that is designed to be scalable up to 512-way multiprocessing; a 64-way machine is currently under construction.

Although the architecture of RP3 does not preclude interactive use, our research interest is motivated by the "single application turn-around" problem rather than the "interactive response time problem." Indeed, with the advent of powerful user workstations, we regard it as being more cost-effective to provide each time-sharing user with his own workstation instead of worrying about how to share a single large machine. Furthermore, the problem of constructing high-performance time-sharing systems has been studied by many researchers and has resulted in competitive products that are readily available, so we do not see this as a fertile research area. Finally, the inherent mismatch between I/O bandwidth and parallel processing capacity means that RP3 is simply not well suited to be a

Supported in part by The Defense Advanced Research Projects Agency under Contract Number N00039-87-C-0122 (Multi-processor System Architecture).

high-capacity time-sharing system. For these reasons we have followed a distributed processing strategy, where RP3 is regarded as a "cycle-server" on a local area network. Workstations on the network are used for "person management" (interacting with the user, following the mouse, keeping the display updated, and so forth) while a large 370 system is used for big compilations, to execute programs that run only under VM/CMS, to store shared data, and to communicate with RP3.

Of course, this approach is not without its drawbacks. Without careful attention to system integration issues, a user may have to maintain separate terminal sessions with RP3, the host 370 system, and the local workstation; worse yet, the user may have to manually move files from system to system in order to complete editing, compilation, linking, and execution of a simple program. Many of these problems can be resolved through use of ubiquitous communication protocols (such as TCP/IP) and clever use of shell scripts that perform file transfer and remote command execution for the user. Our current system environment depends heavily on such mechanisms.

A nicer environment can be based on the AIX/370 "Transparent Computing Facility" (a product-level implementation of Locus [2]). The AIX TCF system has the advantage that it presents a "seamless" system interface to the user; the user need not be aware of the location of a file or process being used. Thus, one might edit a file on RP3 using an editor that runs on the workstation, or interact with a program running on RP3 as if it were running on the workstation.

Another aspect of the RP3 computing environment is the set of system services that programs running on RP3 use. By and large, these system services are identical to those of BSD 4.3 UNIX. But the system interface must be extended to allow users to exploit the special features of RP3. For example, in order to obtain maximum performance, the user must be able to specify those portions of virtual storage should be backed by storage local to this processor and which portions should be backed by global (interleaved) storage. Similarly, the user must be able to specify cacheability of various portions of the program's address space. New system calls must be provided to allow the user to specify these conditions.

Fundamental changes are required in processor allocation and scheduling. The goal of parallel processing is to apply multiple processors against a single user application. Under UNIX, this is often done by creating multiple UNIX processes (typically one process per physical processor in the system) and having each process link to a common shared memory region. This has been a successful approach for many research and commercial implementations. It works acceptably well for the single job environment (although there may be problems related to ownership of resources and sharing of automatic or stack variables) but it can break down badly in the multiple job case.

The problem is that the UNIX scheduler is still scheduling at the process level. It does not understand that a several of these processes have been grouped together by the user and that the user is expecting these processes to be scheduled at the same time. If the scheduler makes inappropriate decisions (for example, suspending a process that holds a lock while at the same time scheduling a process that is in a busy wait loop for the lock), the actual performance of the program can degrade dramatically.

We solve this problem on RP3 by creating a higher level construct called a *family*. The family construct allows the user to specify which processes or threads comprise a particular parallel program and to identify the number of processors that this collection of processes needs in order to run efficiently. The family scheduler then ensures that none of the members of the family will be scheduled unless this minimum number of processors is available. Conversely, the scheduler guarantees not to deschedule a member of the family unless all of the processors are removed from the family. This allows the user to write code that depends on the fact that a specified number of processors will be available whenever the program is running; the user can then plan how to allocate these processors in the program in an efficient a way as possible.

In the following sections of this paper we discuss first the hardware environment of RP3, then the software environment, and finally extensions to the system interface that have been made necessary by the nature of RP3. We will then describe the current computing environment and the integrated system environment based on AIX TCF that is our eventual goal.

2. RP3 Hardware Environment

RP3 consists of 64 *processor memory elements* or PME's; Figure 1 shows the components of the PME. Each PME consists of:

ROMP This is a 32 bit RISC processor; the same processor is used in the IBM RT System.

FPU The floating point unit. An asynchronous, single and double precision floating point unit using S/370 floating point format. It supports scalar and vector commands.

IOI The I/O interface device. The IOI provides an attachment to one of eight S/370 channels; each channel is shared among 8 PME's.

MMU The memory management unit. The MMU supports a typical segment and page table address translation mechanism and includes a 64 entry by 2 set translation lookaside buffer. RP3 uses a 16 kilobyte page size.

Cache A 32 kilobyte, two-way, set associative, one-cycle cache.

MC The memory controller.

Mem Memory module. Note that all memory in RP3 is packaged with the processor. By software convention, a portion of this memory is reserved for *local* storage, the remainder is allocated for use by the entire system as *global* storage.

NI The network interface. The network interface examines each request issued by the processor and determines whether the request is for the local memory (in which case it is passed to the MC), or if it is a request for global memory (in which case it is passed to the SI). A memory address is a local reference if the top 9 bits of the absolute address match the processor number of the current procesor. Otherwise it is a global address and the top 9 bits of the address specify the remote PME.

SI Switch interface. The switch interface connects the PME to the RP3 interconnection network.

PM The performance measurement device. A small number of hardware specific counters are contained in this device. Items measured by this device include: instructions completed, cache hits and misses, and sampled values for the interconnection network response time.

The RP3 hardware does not provide any mechanism for keeping caches coherent between PME's; cache coherency must be supported in software. The cache is visible to application code in the sense that instructions to invalidate portions of or the entire cache are provided in user mode. Additionally, the segment and page tables include cacheability information. The result is that ranges of virtual addresses (on a page boundary) can be specified as cacheable or non-cacheable storage.

Figure 2 shows the overall machine organization of RP3. In this diagram we distinguish between *local* and *global* storage. Local storage is not private storage, it is merely storage that is allocated in one storage module. Local storage is accessible from any processor in the machine. However, there is a performance benefit in accessing local storage from the PME where the storage resides. (If access to the cache is one cycle, then access to local storage requires 10 cycles and access to global storage requires 15 cycles provided the network is idle. Network contention can significantly increase the cost of access to global storage.)

Now if local storage were the only type of storage supported in RP3, potential memory bottlenecks could develop. To avoid this, RP3 supports interleaved storage. Addresses for interleaved storage undergo an additional transformation after virtual to real address translation. The interleaving transformation exchanges bits in the low and high order portions of the real address. (See Figure 3.) Since the high order bits of the address specify the processor number, the effect of the interleaving transformation is to spread accesses to an interleaved page across storage modules in the system. The number of modules used is specified by the interleave amount in the page table; convention is to use all of the modules in the system to support interleaved storage.

The interleaving transformation is applied after a one-to-one hashing transformation. The hashing transformation randomizes sequential storage references as an additional technique to minimize memory conflicts.

Normally, all data used by more than one processor is stored in interleaved storage. For this reason, local storage is also referred to as *sequential* storage and global storage is referred to as *interleaved* storage.

RP3 supports the Fetch and Add operation (as well as Fetch and Or, Fetch and And, etc.) as the basic synchronization primitive. *Fetch and Add* is defined as [9]: *F&Add(location,value)*. An atomic operation that returns the contents of 'location' and then increments the contents of the location by 'value'.

Further details of the RP3 PME and system organization can be found in [3,4].

The RP3 Complex. RP3 is part of a unified group of machines that we call the *RP3 Complex*. Figure 4 shows the major components of this complex. The complex consists of:

- The parallel machine itself, RP3.
- A number of workstations, at present mostly RT systems. PS/2 Model 80's running AIX PS/2 will be added as this system becomes available.
- A large S/370 belonging to the RP3 project; this system runs VM/CMS.
- A local area net connecting the components of the complex.

RP3 and the S/370 are channel connected. Disk I/O is supported through the S/370; the S/370 can be regarded as a back-end I/O processor for the RP3.

Workstations with APA displays were selected as the primary user interface to RP3. The 3270 data stream protocols of terminals on the S/370 were regarded as too confining for the types of user interfaces that were planned for RP3.

3. RP3 Software Environment

Early in the RP3 project, it was decided that RP3 would run a version of BSD UNIX. The reasons for this were relatively straightforward:

- Source code for UNIX was readily available.
- The system was thought to be relatively easy to port.
- Many people are familiar with the UNIX programming interface; a somewhat smaller group is familiar with UNIX internals.
- BSD 4.2/4.3 UNIX is in common use at many Universities. (One of the goals of RP3 was to encourage collaboration with University researchers in the field of parallel processing.)
- UNIX has features (pipes, fork) similar to those required for parallel processing.

Of course, the BSD 4.x systems are not perfect for our use. As delivered, the system is uniprocessor structured, it does not support shared memory between processes (although a vestigial MMAP system call is included), it uses VAX specific virtual memory management, and the only way to create a new process is to also create a new address space (that is, there is no notion of lightweight processes in the system).

For a project in highly parallel operating systems, it was crucial that the multiprocessor limitation of UNIX be quickly overcome. Also, although 7th Edition UNIX from Bell Laboratories might have been relatively easy to port, the BSD 4.x systems we were looking at had grown to the point where a porting effort was non-trivial. Shared memory between processes was crucial in supporting certain models of parallel programming, and because RP3 was designed with fine grained parallel processing in mind, it was important to exploit light-weight tasking as efficiently as possible.

For these reasons, we selected Mach [1] as the operating system for RP3. The features that Mach provided us were:

- Binary compatibility with BSD 4.3 on systems that supported both Mach and BSD 4.3.

- Shared memory between processes.
- Lightweight processes (threads).
- Multiprocessor capable
  It was running MP on VAXen at CMU.
- It was already running on the RT system.
  RP3 uses the same processor as the RT.
- Easier to port than BSD 4.3
  Clear machine independent/dependent split
  Machine independent VM management

Overall, we have found this choice to be of great benefit to our project.

3.1. Remote Access to VM/CMS Tools

However, the operating system is only part of the software environment. Without good compilers and other tools, the machine cannot be programmed efficiently. Here the choice of Mach as the operating system for RP3 has been less auspicious. The best internally available compiler for the ROMP processor is the PL.8 compiler developed by the 801 group in Yorktown. But PL.8 runs under VM/CMS. While in principle one should be able to get the PL.8 compiler to run under Mach, this is an effort that diverts us from our interest in parallel systems and applications. We therefore decided to continue to use the PL.8 compiler under VM/CMS and to convert the output module format to that used by Mach. Similarly, tools for automatic source-code parallelization (such as PTRAN [6,7]) run only under VM/CMS, and there does not appear to be any effective path to get this large PL/I program to run under Mach. Remote execution of these VM/CMS tools is a requirement of the RP3 Computing Environment.

Access to the VM/CMS tools is via ONCMS an internal IBM tool that establishes a UNIX to CMS bridge. ONCMS uses TCP/IP to establish a connection to a VM/CMS userid and then allows the user to execute CMS commands (much as *rsh* allows a user to execute a UNIX command on a remote machine). Shell scripts on the workstation are used to hide (as much as possible) the fact that a particular command has been executed remotely. However, since the remote command typically executes in the requester's VM/CMS userid, the user must know that he cannot be logged on to VM/CMS at the same time that he is issuing the remote service commands. This can be fixed by having an additional userid dedicated for this purpose or creating shared service machines to execute the remote commands; at present we are not planning to do this. Instead we would like to migrate to a more flexible interaction with the S/370 host via way of host based UNIX systems.

3.2. S/370 UNIX Systems in the RP3 Complex

Another way to get access to the computing resources available on the RP3 project's host 370 is to run a S/370 UNIX. At the present time, we are running a pre-release version of IBM's UNIX, AIX, on the S/370.

The AIX system is being used for some production length graphics runs, and for experimentation with the AIX Transparent Computing Facility, or TCF. The goal of the experimentation is to see if AIX and TCF can form the basis for a "seamless system" that can run across all components of the RP3 Complex. The advantages of using this in the RP3 complex are discussed further below.

3.3. Parallel Fortran Support on RP3

At present, parallel Fortran support on RP3 is provided using EPEX [5]. EPEX is a Fortran extension (implemented via a preprocessor) that allows the user to specify *DO*-Loops that can be executed in parallel (that is *DO*-Loops that have no intra-loop dependencies). EPEX was originally developed on VM/370. Under VM, multiple virtual machines were linked to a single copy of a shared data space. Identical copies of the program were loaded into each virtual machine and they then cooperated in accessing the shared data. In essence this approach allows the construction of a virtual multiprocessor under VM.

At present there are a large number of applications written for EPEX/VM, and it was a requirement that we be able to execute these applications on RP3. Figure 5 shows how an EPEX application runs under Mach on the RT or on RP3. The basic idea is to use the Mach *vm_inherit* call to specify which pages of storage in the parent are to be shared among all child processes, and then to create one child process for each processor required by the application. The only difference between EPEX/RT and EPEX/RP3 is the synchronization primitive used: the RT uses *Test and Set Halfword* while RP3 uses *Fetch and Add*. Currently, users can develop and test EPEX applications under Mach/RT and prepare them for execution on RP3 by relinking the object module.

Eventually, we hope to include automatic parallelization of Fortran as a part of the RP3 Computing Environment. The plan is to use the PTRAN source-level restructuring system [6,7] to support automatic parallelization of code for RP3.

3.4. Software Environment Summary

In summary, the initial software for the RP3 Complex consists of

- Mach on the RT systems
- VM/CMS and AIX on the S/370
- A customized version of Mach on RP3
- A version of EPEX/Fortran that runs under Mach/RT and Mach/RP3

4. Mach Enhancements for Parallel Processing

As mentioned before, Mach is an MP capable version of BSD UNIX. However, as obtained from CMU, Mach did not fully meet our requirements for parallel processing. In particular, although it was already multiprocessor structured, it was not yet suited for RP3 because:

- Mach did not run on MP ROMP machines because none exist outside of IBM.
- Mach did not perform job level scheduling (the dispatcher was still scheduling processes instead of entire jobs or families.)
- Mach did not solve the RP3 local memory problem.
- Mach used serial rather than parallel queue manipulation algorithms.
- Mach did not support the RP3 cache or floating point unit.
- All I/O in Mach is initiated from a single processor.

These problems were solved as discussed below.

4.1. Making Mach MP on RP3

Although the machine independent code in Mach is multiprocessor (MP) structured, the version of Mach for the RT system is not MP capable. For example, the number of processors is set to one in the RT configuration file and the calls to set and clear locks are eliminated during the preprocessing phase of C-compilation. Additionally, the machine dependent code for the RT was not MP structured.

We fixed these problems by rewriting the machine dependent code from the RT and basing the pmap module for RP3 (this is the machine dependent half of the virtual memory management code) on that of the VAX. Although RP3 uses the same processor as the RT, the RP3 memory management unit is actually more similar to that of the VAX than it is to that of the RT. Thus all of the changes required to get Mach running MP on RP3 were contained within the machine dependent code.

4.2. Job-level Scheduling in Mach

The implementation of EPEX under Mach (as shown in Figure 5), is typical of how parallel programs have been executed under UNIX. This has been a successful model of execution for many multiprocessor UNIX systems. However, there are some problems with this approach:

Private portions of address spaces. Each of the child processes has a private portion of the address space. Data stored in that portion of the address space is not accessible to other processes in the program. Since the program stack is normally in the private area, this means that data stored in automatic (local) variables within the program cannot be shared between tasks. Additionally, the presence of such private data within a particular process' address space can make that process special in the sense that it is the only process that can complete a particular portion of the job. It may therefore be impossible for the operating system to remove a processor from the program without causing the parallel program to deadlock or otherwise fail.

Local resource ownership. Because a process is the unit of resource ownership, resources may not be shareable between different parts of the parallel program. For example, if one of the child processes opens a file, this file is not accessible from any of the other processes. Of course, this problem can be fixed by opening the file in the parent before the child processes are created, but this may not be possible in all cases. The result of this limitation is that one often must designate a particular process to be the input/output process and have all other processes branch around the I/O statements. While this is the approach used in EPEX/VM, it may introduce an unacceptable serial bottleneck in a highly parallel program.

Processor allocation requires address space creation. This can make changing the number of processors allocated to a particular program prohibitively expensive. This is not a problem in the single program execution case, but in the multiple job parallel processing case, processor reallocation is occasionally a requirement. Furthermore, in the EPEX model, it is not clear how to organize the code so that a user can take advantage of the fact that an additional processor has become available or recover from the fact that a processor has been removed (due to the private data problem mentioned above). Additionally, startup overhead can become

significant when many processors are involved and large numbers of address spaces must be created.

4.2.1. Mach's Task/Thread Model

In the Mach operating system [1,8], the process construct has been split into the primitives *task* and *thread*.

A *task* is the unit of resource ownership. It includes a large and potentially sparse address space and owns a UNIX process id. It also can be granted communication port rights; these are software capabilities that allow programs to send and receive data.

A *thread* is the unit of processor execution and can be thought of as flow of control or a lightweight process. A UNIX process is emulated under Mach by creating a task and starting a single thread within that task. But Mach also allows the creation of multiple threads within a single task and in a multiprocessor environment each of these threads may be executing on a separate processor. Thus through the use of the Mach task/thread concepts one can:

- have multiple processors executing in the same address space
- decouple processor allocation from address space creation

The first item eliminates the problems associated with having private portions of the address space (albeit at the risk of losing protection for those regions against modification by other processes) and requires the creation of only one address space regardless of the number of processors involved. The second item allows us to add or remove processors from a particular program and allows a simple method of organizing the program so that it readily adapt to such a change.

4.2.2. Programming using the Task/Thread Model on RP3

Instead of the EPEX model of computation, we propose that new applications on RP3 be written to the Task/Thread Model (or TTM).[1] The TTM has the following characteristics:

- Each job runs in single address space.
- There may be multiple threads per address space.
- There is a local work queue defined by the user.
- Each thread selects work to do by removing an entry from the work queue, executing that request to completion and then returning to the dispatch queue for the next work item.
- Execution of a work item can cause creation of new work items to be placed on the local dispatch queue.
- New processors are introduced into the address space by creation of new threads. These threads begin execution by selecting a work item from the work queue.

[1] Indeed, the principal reason for supporting the EPEX model on RP3 at all was to allow the existing base of EPEX/VM applications to be ported to RP3.

- Processors may be removed from the address space by suspending execution of a thread, saving the local context of the thread in the work queue, and deactivating the thread within the address space.

This approach allows the user to specify how processors are scheduled at the level of the user's definition of the problem. For example, in a parallel Fortran environment, a work item may be execution of a single *Do*-Loop iteration, and selection of the next work item may be as simple as grabbing the next loop index value from a global variable.

If there are multiple jobs executing at the same time, a "global processor allocation" component of the operating system is responsible for balancing processor allocation among the jobs. Because allocating a processor is still an expensive operation, we expect global processor allocation decisions to be made on a medium term basis (minutes, rather than seconds) and that once a processor is allocated to a job it will remain allocated to that job for a reasonable period of time.

Figure 6 shows a diagram of the TTM model in the multiple-job parallel processing case. Note that we do not envision running a large number of jobs in parallel on RP3, but rather enough separate jobs to keep the machine fully utilized. We expect that something like 4 to 8 jobs might be the maximum number we would allow at any one time.

4.2.3. Processor Allocation in RP3

Thus far our description of the TTM model has not described how the global processor allocation algorithm works. In general, this is a policy issue that is one of the subjects of our current research. However, we have an initial processor allocation policy that is guided by the concept of *job-level scheduling*. The following discussion describes the policy we plan to support in RP3.

The concept of job-level scheduling is inspired by the working-set philosophy of managing virtual memory. Basically, the idea is that each job in the system has a minimum number of processors that it requires in order to execute properly. If that number of processors is not available, then the (parallel) job should not run. Similarly, the system agrees not to remove a processor from an executing job if that removal would reduce the number of processors below this threshold. Additionally, a job is allocated processors on a long term basis, and the user can depend on this fact when writing the application program.

Eventually, we believe it may be possible to dynamically determine the correct "processor working-set" of a job, much as paging systems estimate the working set of an executing program. But this is a research issue and at present we are depending on the user to specify the number of processors that the job requires.

To support the job-level scheduling policy, we are adding the *family* primitive to the Mach system. A family is

- a group of threads dispatched as a unit
- the largest scheduling unit recognized by the operating system
- an entity that runs under control of the *family scheduler*

The intent of the family is that each parallel application consists of a single family. Of course, complex applications with multiple independent subunits might consist

of several families, but for our present discussion we will assume that each application is composed of only one family.

Once a family has been created, a user may request that processors be allocated to that family.[2] The family scheduler will ensure that the members of the family will be scheduled only if the specified number of processors are available. Figure 7 shows an example of how a family consisting of five processes would be created within Mach/RP3. The parent process creates the family using the *family_create* system call and then specifies that five processors are required by this family using the *proc_alloc* system call. (The other arguments to *proc_alloc* allow the user to specify a minimum and maximum number of processors to be allocated, and whether or not the user wishes to be suspended until the processor allocation can be satisfied. If the user specifies *NO_BLOCK*, then if the processor requirement cannot be met, the user can retry the request with a different number of processors.)

Additionally a user may wish to specify that processors owned by the family be dedicated to particular threads in the family. In Mach/RP3, this is done using the *thread_bind* system call. This system call allows the user to specify a thread to be bound to a processor, but not which processor is to be allocated. Instead, one of the processors belonging to the family is chosen by the system and bound to the thread. The reason for doing this is that it eliminates the problem of naming processors and dealing with error situations if the user tries to allocate or release a processor that does not presently belong to the family. (Unbound threads share the unbound processors in the system and it is an error for the user to create more bound threads than there are allocated processors in the family.)

Figure 8 shows how each process in the family is allocated a processor using this mechanism. Recall that in Mach, each UNIX process is emulated by using a single thread within a task, so while the argument of *thread_bind* is a thread ID, if there is only one thread defined in the task, then binding that thread is equivalent to binding a processor to the process consisting of that task and thread.

A family is defined to consist of the threads in the current task and all threads in tasks that are offsprings of the current task. In a complex application, a user can define a new family at any point in the hierarchy. Figure 9 shows how a new family can be created. The result of the new family definition is that tasks 0, 1, 2 and 4 are dispatched as a unit and tasks 3, 5, and 6 are dispatched as a second unit. They both may run at the same time if there are sufficient processors available, or they may alternate execution if there is a processor shortage.

4.3. Supporting Local Memory in Mach/RP3

Once a thread has been bound to a processor, the user may request that specified virtual address ranges be backed by storage local to that processor. (By default, virtual storage is backed by interleaved, global real storage.) However, in keeping with the philosophy of not using processor ID's in the system interface, the

[2] In the EPEX model, a family may consist of several tasks. Thus, it is inappropriate for the task to be the unit of processor ownership.

way this is done is to request that the storage be local to a particular (bound) thread.

If the virtual storage exists at the time of the local memory request, then the content of that storage is copied from global to local memory as part of the execution of the system call. In general, we expect that the user will make the local memory request sufficiently early in the execution of the program that no data will actually have been loaded into the virtual storage range before the local memory request is issued.

In our current implementation, the number of local storage pages is fixed at system boot time. It is therefore possible that the user may issue requests for more locally backed storage than there are local storage pages. Our current plan is to treat this as an error situation and not attempt to page between local and global storage. The rationale for this is that local storage is a performance optimization done by the user. If the operating system attempts to page this storage without the user's knowledge, it is unlikely that any performance improvement will result.

4.4. Control of Cacheability

User control of cacheability is handled in a similar way. Machine operations to invalidate the cache are not privileged operations and can be performed by the user at any time. Marking a portion of virtual storage as cacheable or non-cacheable requires changing the page and (perhaps) segment tables for that portion of the address space. A modified version of the Mach *vm_protect* system call is used to allow the user to specify the cacheabiliy attributes of a virtual storage range. The system makes no attempt to determine if the user is properly using cacheable storage. (Since in RP3, caches on different PME's are not kept consistent, using cacheable storage to hold read/write shared data may cause unpredictable results.) Proper use of cacheable storage is regarded as a user or compiler responsibility.

4.5. Extensions to a.out Format

The mechanisms specified above allow an executing program to allocate processors, identify local versus global storage, and indicate cacheability attributes for portions of the user's virtual address space. However, a compiler may wish to specify these attributes directly, as part of the load module format. Eventually, we plan to support an extended version of the System V COFF format that includes this type of information. Since COFF already supports multiple segment types above and beyond those of traditional UNIX, special segment types can be defined for local, cacheable, or interleaved storage. The processor requirement for the program could also be encoded in the module format much as the memory requirement is now specified there. The operating system scheduler could use this information to determine whether it is appropriate to begin execution of the module based on the module's processor requirement and the current number of free processors.

4.6. Parallel Queue Management in Mach/RP3

Traditional estimates of the maximum speedup possible for a parallel program have been based on Amdahl's law: *Compute time = parallel part + serial part*. Now

if we assume that a particular program is 99 per cent parallel and only 1 per cent serial, and if we assume that the parallel portion exhibits linear speedup, then if 99 processors are assigned to the problem we see that the program spends half of its time in the serial section of the code. Thus to achieve maximum speedup it is crucial that serial sections of the code be as short as possible.

A number of researchers at New York University [9] have developed serialization free algorithms for such common operating system functions as: queue management, stack manipulation, storage allocation and the like. These algorithms use the primitive *Fetch and Add* instead of instructions such as *Test and Set* or *Compare and Swap*. RP3 supports the *Fetch and Add* instruction and these algorithms can be used in RP3/Mach.

Given our style of programming the machine (that is, no interactive time sharing, job-level scheduling and the like), it remains a research question as to whether these algorithms are required in RP3/Mach. Indeed, a much more significant bottleneck currently exists in Mach; this is described in the next section. Nonetheless, it is apparent that for certain workloads, execution of the system dispatcher may become a significant serial bottleneck. To evaluate the significance of this problem, we have developed a version of the Mach scheduler using Fetch and Add. Performance evaluation of this scheduler is a current research topic.

Even if the system scheduler is not a serial bottleneck, such bottlenecks may exist in the user code. For example, in the TTM model of programming, the threads repeatedly select a work item from the work queue, and this operation may become a serial bottleneck. To avoid this, we are also creating subroutine libraries to perform standard operations (such as parallel queue manipulation) that use the Fetch and Add primitives. These libraries will be made available to users so that they may be linked into application code.

4.7. Elimination of Serial Bottlenecks within Mach

Even if all serial locks within the Mach kernel were replaced by serialization free algorithms, current versions of Mach still contain significant serial bottlenecks due to the master-slave implementation of the UNIX compatibility portions of the kernel. In Mach Release 2.0, for example, all UNIX system calls are executed by a designated "master" processor (typically processor zero). System calls issued by code running on another processor are serviced by suspending the requesting task and rescheduling it on the master processor. Similarly, all I/O is initiated only from the master processor. This avoids having to modify the existing UNIX code to make it MP capable.

The plan for Mach is to remove all UNIX code from the kernel, placing this function instead in transparent emulation libraries loaded with the user's code. Hence, the Mach research team has not emphasized parallelization of the UNIX portion of the code. (The code written to support the Mach functions is symmetric code; it can run on any available processor.)

This problem has been partially solved in an experimental version of the Mach kernel called the "non-buffer cache" or NBC kernel. In the NBC kernel, key system calls (such as open, close, read, and write) may execute on any processor, and support allowing I/O to be initiated from any processor is mostly complete.

We are currently evaluating the prospects of migrating RP3/Mach to an NBC base. Alternatively, we can wait for Mach 3.0, which will provide UNIX compatibility through the transparent emulation library. At present, this bottleneck does not appear to be a serious one, since the tendency of our application programmers is to avoid system calls altogether once the application program environment has been created.

5. RP3 System Image

As mentioned in the Introduction, we do not see RP3 as a general purpose time-sharing host, but rather as a cycle-server on a local area network. Users interact with RP3 principally via workstations on the network. In this section we describe the current software environment in more detail and discuss how this environment may evolve over time.

5.1. RP3 Cooperative System Image

At present we support a "cooperative" system image. That is, the various machines in the RP3 complex cooperate in the preparation and execution of a user program and the user is likely aware that more than one machine is involved in this process. Normally, the machines involved are the user's workstation and RP3.

A typical edit, compile, run sequence of an EPEX/Fortran program would be performed as follows:

- The user edits code using the workstation editor (*vi* or *emacs*) and creates a Fortran program that includes the EPEX language extensions.
- Using the EPEX preprocessor, a parallel Fortran program is then produced and compiled using F77.
- The program is linked with the EPEX/RT library and tested (in uniprocessor mode) on the workstation.
- Once it works on the RT, the program is relinked using the EPEX/RP3 version of the library.
- The user uses *rsh*, *rlogin*, or *telnet* to begin executing the program on RP3.

The version of Mach on the RT and the one on RP3 are kept as similar as possible. For example, new system calls for processor allocation exist on both Mach/RP3 and and our versions of Mach/RT. On the RT, many of the system calls are merely stubs, but the system call handler at least checks arguments for validity before returning. However, it is impossible to run the same module on RP3 as on the RT since the latter does not support Fetch and Add.

Access to files is simplified through the use of the CMU remote file system included in Mach. Through use of this remote file system, programs need not be moved to RP3 before execution, but instead can be executed in place from RP3. Depending on the amount of data involved, data files can be treated similarly, provided path names are changed appropriately. Large files need to be copied to RP3 to allow use of RP3 native I/O to access the files instead of using the LAN. Eventually we see the CMU remote file system being replaced by NFS or the Locus replicated file system.

5.2. RP3 Seamless System Image

Clearly the "cooperative" system image has little to distinguish it from countless similar environments composed of workstations, hosts and LANs. However, we envision replacing this cooperative system image with a "seamless" system image based on the AIX Transparent Computing Facility. The intent is that the user need not be aware of machine boundaries in the preparation and execution of a program.

AIX Transparent Computing Facility. The AIX Transparent Computing Facility (or TCF) is a product level implementation of Locus [2] announced for IBM/370, IBM/370-XA and PS/2 Model 80 machines.

As discussed in [2], the key concept of Locus is *network transparency*. The idea is to provide a level of network support (within the operating system) that makes the network virtually invisible to users and application programs. As much as possible, Locus attempts to make a network of computers as easy to use as a single computer. The AIX TCF implementation updates Locus by supporting the AIX Family Interface (an extension of System V.2 that is also supported on AIX/RT).

Features of AIX TCF are that

- It supports a single distributed file system that is location transparent. This means that the path name of a file is the same regardless of which machine the file is accessed from, with certain exceptions (/tmp being replicated on each machine, for example).
- The file system includes file replication with automatic update. Files may be replicated to improve performance or reliability.
- AIX TCF supports heterogeneous machines as was done in Locus. AIX/370 and AIX PS/2 can coexist in the same TCF cluster. AIX handles byte order and certain other conversions automatically when data is passed between machines of different types. The machine type is contained in the module header, and the system supports storing modules for different machines under the same program name. When the program is invoked on a machine of a particular type, the system selects the module matching that machine type (if it exists).
- The system supports transparent remote execution. This is either explicit (by user request) or implicit (because the user invoked a program on one type of machine, and the system was only able to find a version of the program for a different machine type). Output from programs executed remotely is indistinguishable from output generated locally.
- The system supports process migration between machines of the same type.

Using AIX TCF in the RP3 Complex. If AIX TCF were available on RP3, then the program development scenario given above could be simplified by:

- Allowing all programs and data to be accessed using the same file names from both RP3 and the users workstation.
- Assuming a new machine type of RP3 is introduced, the user would need not use *rlogin* or *rsh* to execute a program on RP3. Instead, once the program has been linked for use on RP3, merely typing the program's name would cause it to be invoked on RP3.

Additional benefits from a system management point of view are that:

- Programs and data can be kept on large, reliable disks on the RP3 S/370 host, instead of on the workstation.
- Large scale compilations could be done on the S/370 host under AIX/370.
- Keystroke intensive interaction can be isolated on the workstation.

In short, each machine in the complex is allowed to perform those functions it is best suited for: RP3 for computation, S/370 for data storage and I/O, and the workstation for interaction with the user.

Our present plans in this area are to explore merging Mach/RP3 and AIX/370, much in the same way the CMU has extended BSD 4.3 into Mach. Whether we can achieve the seamless system image we have described remains a research question.

6. Current Status

As of August 1988, our status is that:

- An 8 processor engineering prototype of RP3 is running.
- Production runs of cards for the 64-processor machine are underway.
- The EPEX environment runs on the RT and under Mach/RP3.
- The "cooperative" system environment is in place.
- The Task/Thread Model is supported using the C-threads package from CMU [10].
- Mach/RP3 is being extended to support
  interleave and local memory
  job-level scheduling
  parallel run queues

Acknowledgments

The components of the RP3 Computing Environment are the work of many people, without whose conscientious efforts the system would not exist. In particular, the Mach port to RP3 is due to Henry Chang, Rajat Datta, and Bryan Rosenburg of the IBM Research Division. The processor allocation implementation is due to Henry Chang. Bryan Rosenburg is responsible for the local and interleaved memory support. Doug Kimelman of IBM Research and Alan Gottlieb of New York University are responsible for the parallel queue implementation in Mach/RP3. Tony Bolmarcich was responsible for the implementation of EPEX for Mach/RT and Mach/RP3. Installation and support of AIX on our project machine has been made possible by Sig Handelman of the IBM Research Division. The original port of Mach to the RP3 simulator was completed by Dan Julin of Carnegie Mellon University while he was a summer student working with the IBM Research Division. The Mach research group at Carnegie Mellon University, led by Professor Richard Rashid, has been very supportive of our work with Mach and their special efforts to make the latest versions of Mach available to us are greatly appreciated. Dan Rencewicz of IBM/TCS has been responsible for obtaining production releases of Mach for the RT system and making them available within IBM.

References

[1] Accetta, Mike, et al., "Mach: A New Kernel Foundation for UNIX Development," in *USENIX Association Proceedings*, Summer, 1986.

[2] Popek, Gerald J. and Bruce Walker. *The LOCUS Distributed System Architecture*, The MIT Press, 1985.

[3] Pfister, G. F. et al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing* , pp. 764-771.

[4] Brantly, W. C., K. P. McAuliffe, and J. Weiss, "RP3 Processor-Memory Element," *Proceedings of the 1985 International Conference on Parallel Processing* , pp. 782-789.

[5] Stone, J. M., et al., "Introduction to the VM/EPEX Fortran Preprocessor," IBM Research Division Technical Report RC 11407, September 30, 1985, P. O. Box 218, Yorktown Heights, New York.

[6] Allen, Frances, et al., "An Overview of the PTRAN Analysis System for Multiprocessing," *Proceedings of the 1987 International Conference on Supercomputing*, Springer-Verlag, Athens, Greece, 1987. To appear in a special issue of the Journal of Parallel and Distributed Computing.

[7] Allen, Frances, et al., "A Framework for Determining Useful Parallelism," *Proceedings of the ACM International Conference on Supercomputing '88*, St Malo, France, July 1988, pp. 207-215.

[8] Tevanian, Avadis, et al., "Mach Threads and the UNIX Kernel: The Battle for Control," in *USENIX Association Proceedings*, Summer, 1987.

[9] Gottlieb, A., B. D. Lubachevsky, and L. Rudoplh, "Coordinating Large Numbers of Processors," *ACM TOPLAS*, January 1982.

[10] Cooper, Eric C. and Richard P. Draves, "C Threads," Draft Technical Report of 2 March 1987, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

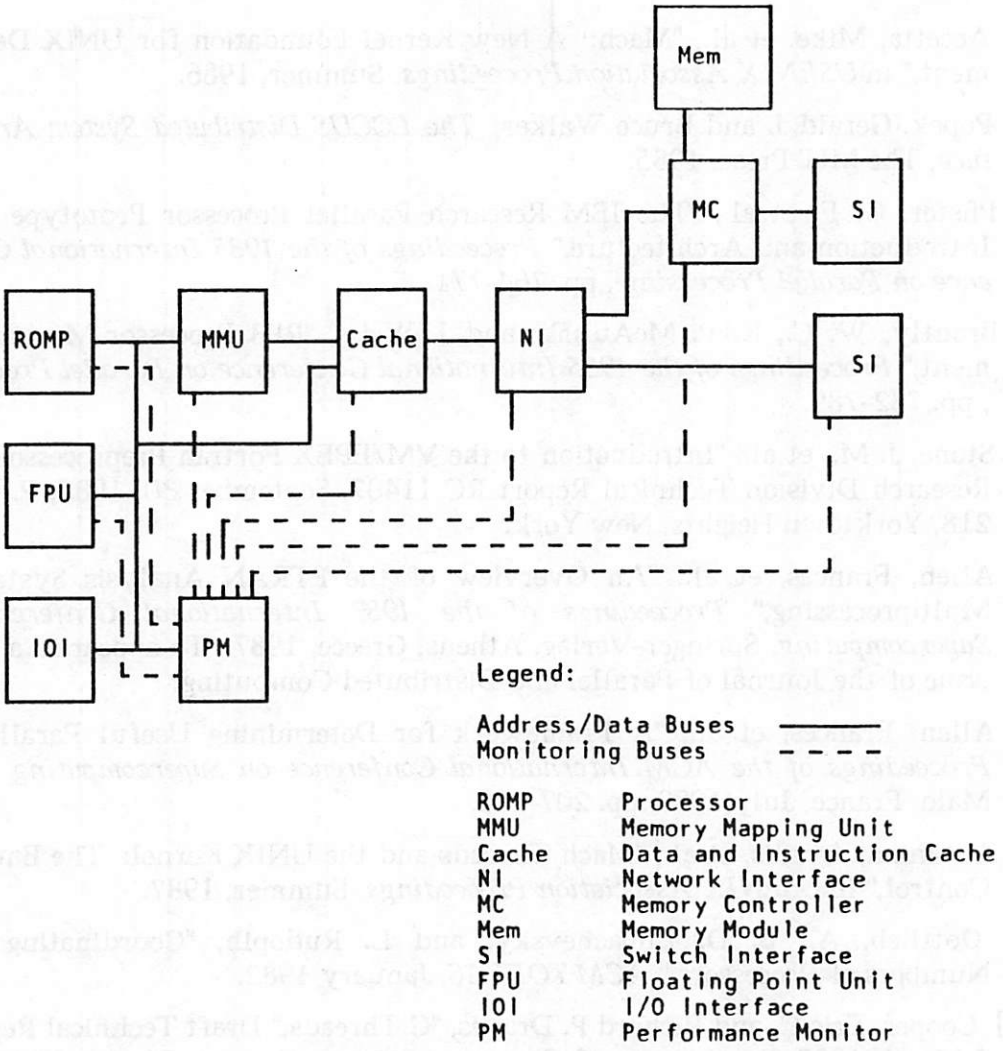


Figure 1: RP3 Processor-Memory Element (PME)

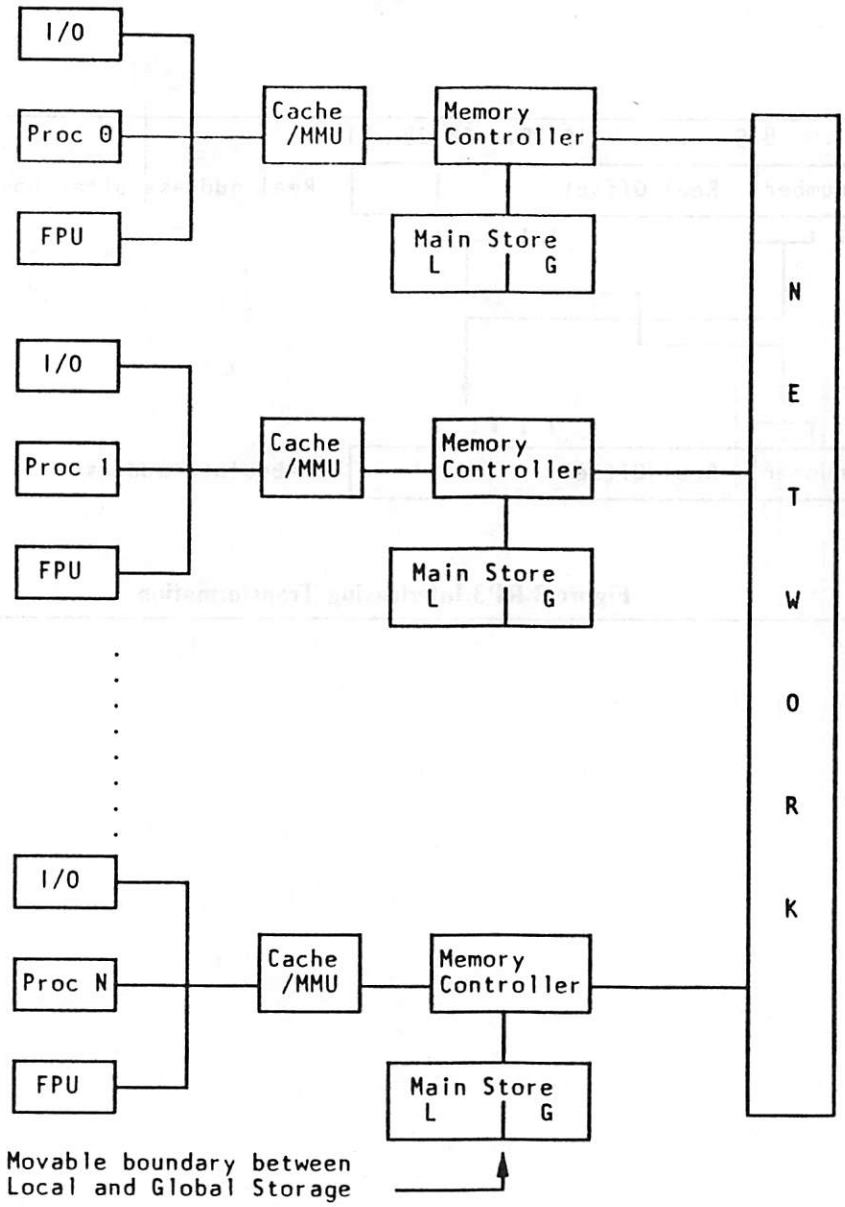


Figure 2: RP3 Machine Organization

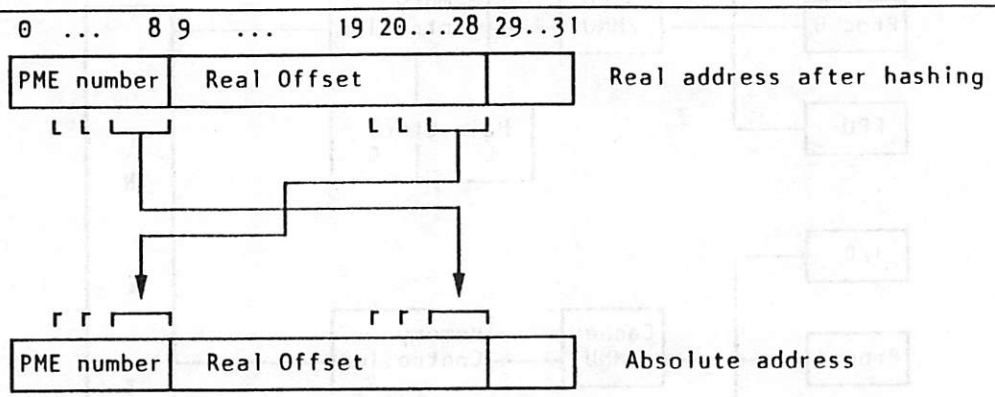


Figure 3 RP3 Interleaving Transformation

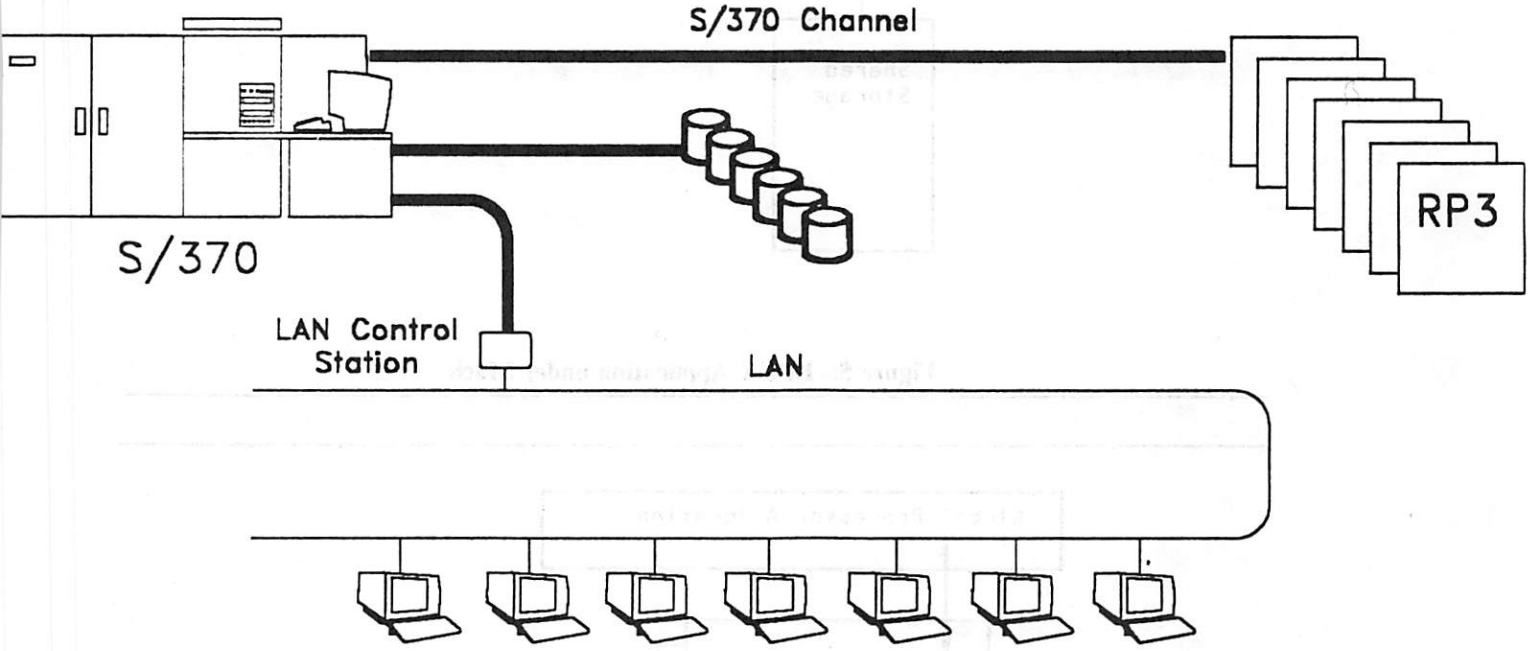


Figure 4: RP3 H/W Complex

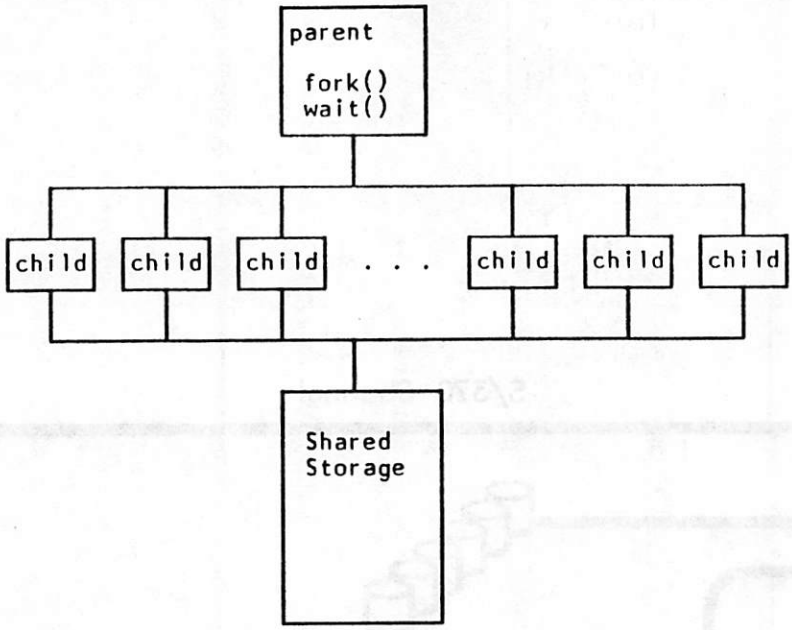


Figure 5: EPEX Application under Mach

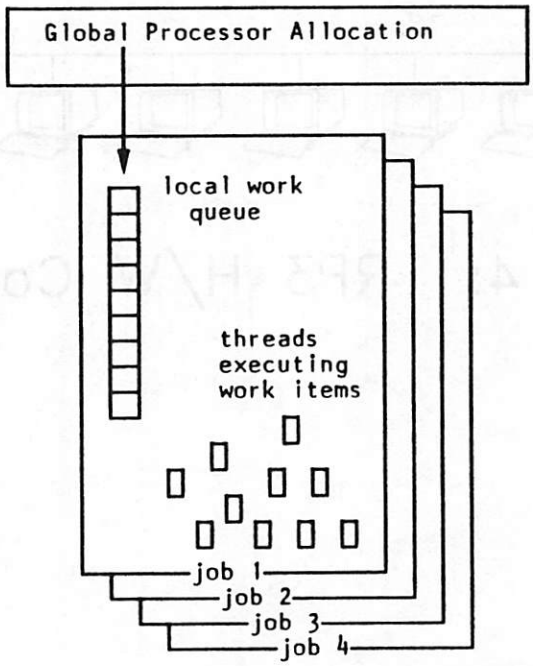


Figure 6: Task/Thread Model in Multiple Job Case

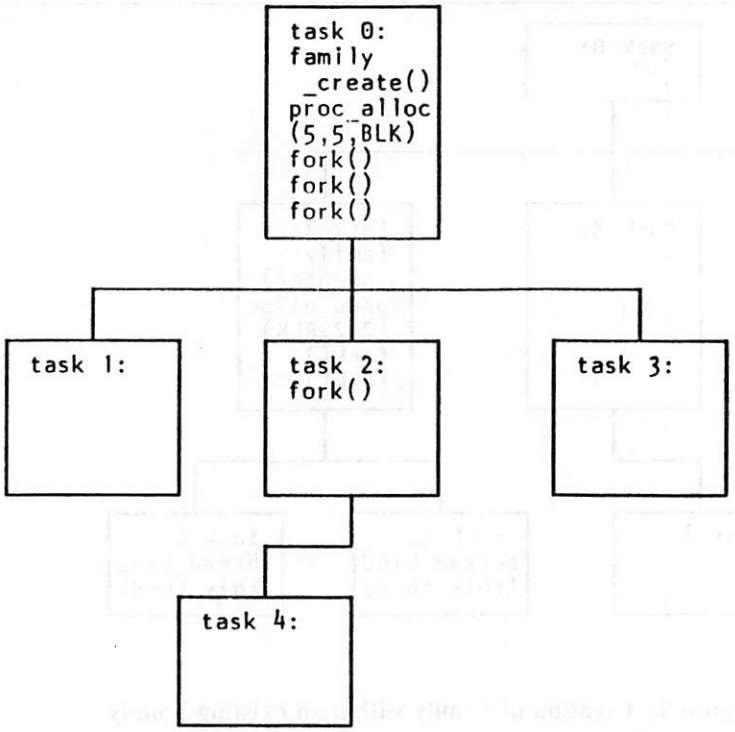


Figure 7: Family Creation in Mach/RP3

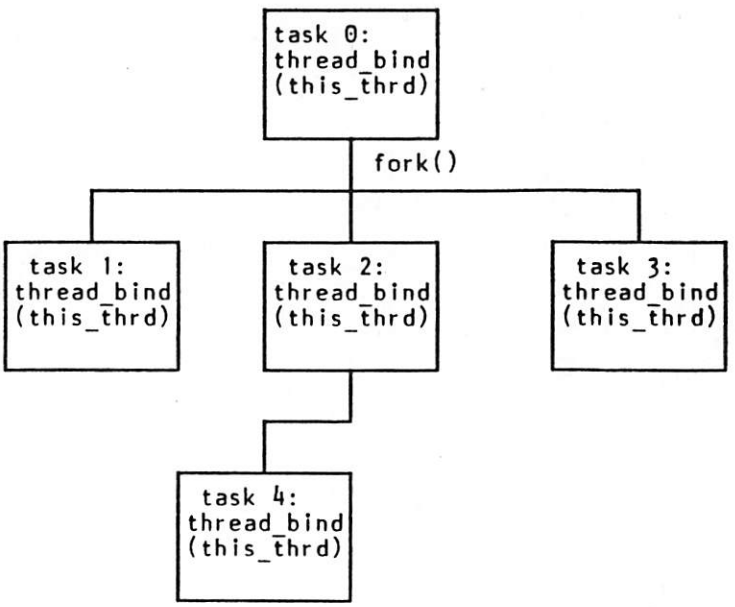


Figure 8: Processor Assignment within a Family

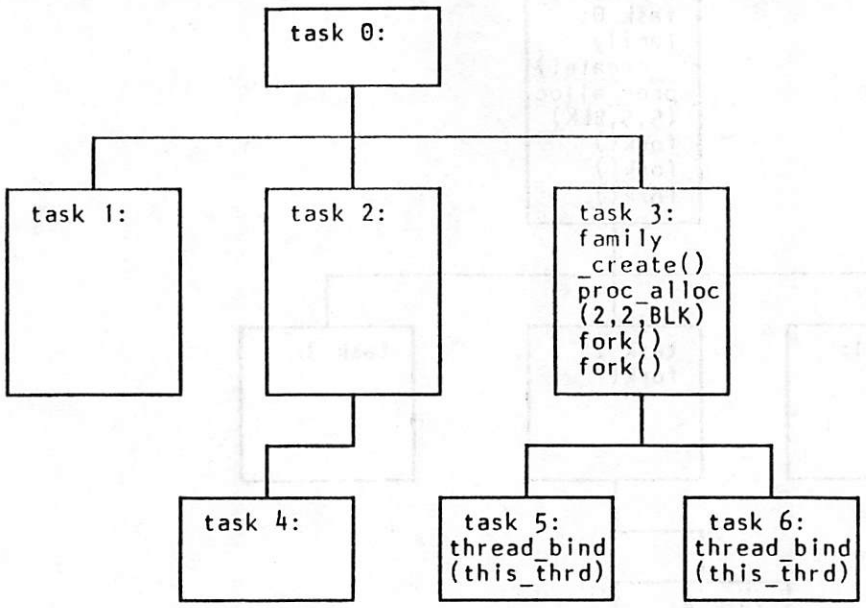


Figure 9: Creation of Family within an existing Family

UNIX and the Connection Machine Operating System

Brewster U. Kahle
William A. Nesheim
Marshall Isman

Thinking Machines Corporation
Cambridge, Massachusetts

ABSTRACT

The UNIX operating system is used on several different computers that comprise the Connection Machine®[1] system. This makes the Connection Machine operating system a simple distributed operating system optimized for large data applications. Common minicomputers are used as front ends for the data processors, with one controlling the disk system, and other computers used as I/O processors. The front end acts as a center of control for the other parts of the system, and is connected to the data processors via an interface on its I/O bus. Programs run on the front end, and instructions are sent to the Connection Machine (CM) processors to read, write, and manipulate data in CM memory. CM file transfers are set up and monitored by the front end and the disk controlling computers while the data is transferred over a high speed I/O bus. Application specific data transfers are handled by other I/O processors over a general purpose interface to the high speed data I/O bus. UNIX is used as the operating system on these components to provide a unified operating system model for the system.

This paper will illustrate how the pieces of the Connection Machine System are integrated to provide a distributed operating system based on UNIX. The structure and the integration of the overall system will be discussed without going into detail on the design and implementation of each. Furthermore, a prototype method of timesharing a massively parallel machine will be outlined. Finally, a list of interesting issues still to be addressed in operating systems for massively parallel computers will be presented.

1. Introduction

The Connection Machine system is a data parallel computer that is designed to run data intensive applications. This system consists of a conventional serial machine as a front end, many tens of thousands of data processors, and a selection of I/O devices and processors. A program executing on the front end commands all the components of the Connection Machine system including the data processors, disk system, and I/O devices. The user program is in many ways similar to a conventional serial machine program using UNIX like devices, but at run time

[1] Connection Machine is a registered trademark of Thinking Machines Corporation.

many interconnected computers are used to perform operating system functions. Controlling these computers requires a distributed operating system. This paper describes the Connection Machine operating system.

The Connection Machine operating system is a hybrid of UNIX and CM specific code that controls the components of the system. Since the CM is quite different from conventional computer systems, its computational model will be briefly described. An overview of the hardware and software components of the system are given, and the interface between UNIX and the various portions of the Connection Machine system are described in some detail. Further work on the Connection Machine operating system is suggested.

2. A Data Parallel Computer System

Two distinct types of parallelism can be found in today's parallel computers. *Control parallel* computers achieve increased performance by taking advantage of parallelism found in the control structure of programs. The Cray X-MP, BBN Butterfly,[2] and CalTech Cosmic Cube[3] are examples of control parallel computers. In these machines, each processor executes a portion of the program. Consequently, each processor must have capabilities comparable to the processor of a serial computer on which the same program could be run.

Data parallel computers achieve increased performance by taking advantage of parallelism found in the data of a problem.[4] The Connection Machine system, DAP,[5] and Massively Parallel Processor[6] are examples of data parallel computers. Data parallel computers consist of a single instruction engine, and thousands of data processors, each having local memory and connected to a communications network over which they may exchange information with other processors. There are two reasons the factors affecting the design of data processors in a data parallel computer are quite different from those affecting the design of the processors in a control parallel computer. First, the control aspects of a program on the data parallel computer may be executed by the instruction engine. This means the data processors are not required to handle instructions, and may instead be optimized for data manipulation. Second, data parallel problems have tens of thousands of data elements which may be operated on simultaneously with minimal interprocessor interaction.

Programming a data parallel computer is more akin to programming serial machines than to programming control parallel machines. Data parallel programs have only one control sequence, and the program executes on one processor, the

---

[2] Bolt Beranek and Newman Inc. *Development of a Butterfly Multiprocessor Test Bed*, Report No. 5872, Quarterly Technical Report No. 1, March 1985.

[3] C. L. Seitz, *The Cosmic Cube*, Communications of the ACM, Vol. 28, No. 1, January 1985

[4] Thinking Machines Corporation, *Introduction to Data Level Parallelism*, Thinking Machines Corporation Technical Report 86.14, April 1986

[5] Flanders, P.M. et al, *Efficient High Speed Computing with the Distributed Array Processor*, High Speed Computer and Algorithm Organization, Academic Press, 1977, pp. 113-127.

[6] Batcher, Kenneth E. (1980). *Design of a Massively Parallel Processor*, IEEE Transactions on Computers, C-29 (9).

front end. Thus, the program is running in a familiar environment with familiar tools. The data resides in Connection Machine memory and is manipulated by the CM data processors, which can access the memory of other CM processors by using a high speed intercommunication network. Similarly the front end can access the CM memory easily and efficiently. A major difference between data parallel and serial code is that iteration over the data objects is unnecessary, as all data objects can be operated on at once.

3. Connection Machine System Components

The Connection Machine system can consist of several front end processors, a dividable block of data processors, DataVault[7] disk units, high speed graphic display systems, and various I/O computers. While many combinations are possible, this section will describe one configuration in order to illustrate the function of each component in the system (see figure 1).

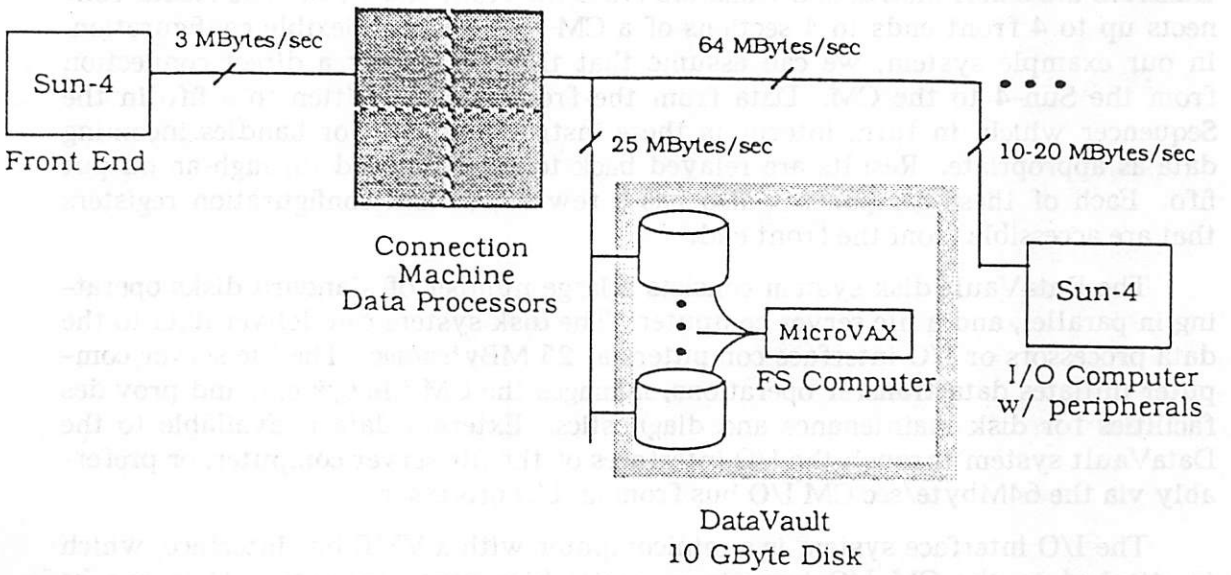


Figure 1: Example Connection Machine System

A Sun Microsystems®[8] Sun-4[9] workstation can be used as a front end to the Connection Machine processors in running user programs. This processor controls resource allocation, performs scalar computation and executes the control portions

[7] DataVault is a tradmark of Thinking Machines Corporation.

[8] Sun Microsystems is a registered trademark of Sun Microsystems, Inc.

[9] Sun-4 is a trademark of Sun Microsystems, Inc.

of Connection Machine programs. No programs are stored or executed in the CM data processors. The user programs run on the front end and issue instructions to the Connection Machine. Data may also be transferred between the front end and the Connection Machine processors via this interface.

The data processors store and process the larger data segments of an application. Each data processor operates on a different piece of data using the same instruction from the controlling front end processor. The data processors are very simple; a typical configuration consists of 16k - 64k processors containing 128 to 512 megabytes of memory.[10] A 64k processor machine performs a 4k by 4k matrix multiply at about 2500MFlops.[11] Each processor can efficiently access the memory of other processors and process the data within its local memory.

The connection between the scalar front ends and the data processors is a front end bus interface *(FEBI)*, a crossbar switch called the *Nexus*, and a *Sequencer* that takes macro-instructions and issues lower level instructions (nano-instructions) to the individual data processors. The FEBI is connected to the system I/O bus of the front end computer. Scalar transfers from the front end to the Sequencer run at approximately 2.5-3.5 MBytes/sec. Direct memory access block transfers are not supported on this interface because the bulk of the transfers are small instruction transfers from the front end CPU. The Nexus connects up to 4 front ends to 4 sections of a CM to allow for flexible configuration. In our example system, we can assume that there is simply a direct connection from the Sun-4 to the CM. Data from the front end is written to a fifo in the Sequencer which, in turn, interprets these instructions calls or handles incoming data as appropriate. Results are relayed back to the front end through an output fifo. Each of these components also has a few status and configuration registers that are accessible from the front end.

The DataVault disk system consists a large number of standard disks operating in parallel, and a file server computer. The disk system can deliver data to the data processors or I/O interface computers at 25 MBytes/sec. The file server computer initiates data transfer operations, manages the CM file system, and provides facilities for disk maintenance and diagnostics. External data is available to the DataVault system through the I/O interfaces of the file server computer, or preferably via the 64Mbyte/sec CM I/O bus from an I/O processor.

The I/O interface system is a minicomputer with a VME bus interface, which is attached to the CM I/O bus via a special high speed interface. Data can be transferred between VME peripherals and the CM data processors or the DataVault under control of the Connection Machine operating system and the I/O interface computer. The speed of the connection depends on the peripherals and I/O processor involved, but 10-20 MBytes can be expected from optimized software and hardware combinations.

---

[10] Thinking Machines Corporation, *The Architecture of the CM-2 Data Processor*, Thinking Machines Corporation Technical Report HA88-01, April 1988

[11] Thinking Machines Corporation, *Model CM-2 Technical Summary*, Thinking Machines Corporation Technical Report HA87-4, April 1987

The resulting Connection Machine system can sustain high enough I/O bandwidths to keep the data processors busy, and have enough front end speed to handle several Connection Machine users. The remainder of the paper discusses the software and operating system tasks performed by these components.

4. The Connection Machine Operating System (CMOS)

The Connection Machine system operating provides many of the features found in a conventional operating system but implements them in a distributed manner. Parts of the CMOS run on each front end computer, the Sequencer, the file server computer and any other I/O interface computers. This section explains where the functions of resource allocation and management are performed. Further details on the components are discussed in later sections.

The CMOS functions generally are layered on top of the local operating system running on each computer within the CM system. Due to the number of different computers involved and the standard nature of the local operating system functions required, it does not make sense for the CMOS to be the native operating system on each computer in the CM system. Instead a common operating system, which is some variant of UNIX, is used on each computer. The front end computer is a Sun-4 running SunOS[12] or a VAX®[13] running ULTRIX[14], the file server computer is a MicroVAX running ULTRIX, and the I/O interface computer can be any VME based UNIX system. The Symbolics[15] Lisp Machine, as a front end, is the only non-UNIX computer that currently can be a part of the Connection Machine system. Each local UNIX is used to provide interprocess communication primitives needed to communicate among the distributed components of the CMOS. The local OS is also used to provide protection, local file storage, and switching between multiple local processes. The only modification to the local UNIX OS is a device driver for the CM specific device on each computer.

The *cmattach* command is used to obtain access to the CM system. This command is analogous to logging in to a UNIX system; once this operation is complete, use of the CM can begin. A *cmfinger* command, analogous to the UNIX command *who*, uses the same UNIX IPC mechanisms to list all current CM users.

Process management and memory management are handled by a combination of the front-end computer and the Sequencer. All policy level decisions are handled on the front ends and some mechanism level functions are implemented by the Sequencer.

The Connection Machine File System (CMFS) provides a UNIX-like hierarchical file system with a programming interface that parallels the UNIX file system call interface. The file server runs on a MicroVax located inside the DataVault parallel disk subsystem. The file server daemon is an ULTRIX user process that takes advantage of UNIX facilities for path name resolution and communication operations to service CMFS operations. Portions of the file system code run on the

---

[12] SunOS is a trademark of Sun Microsystems, Inc.

[13] VAX is a registered trademark of Digital Equipment Corporation.

[14] ULTRIX is a trademark of Digital Equipment Corporation.

[15] Symbolics is a trademark of Symbolics, Inc.

front end computer, portions on the CM Sequencer, and portions on the file server computer. The front end and file server computers use UNIX IPC for communicating control information, and the high speed parallel CM I/O data bus is used for data transfers.

An I/O interface computer can also participate as one end of a file system transaction. This computer can act as a client of the DataVault, for transferring data to or from the disk subsystem, or as a server for the data needs of the CM data processors. This computer is interfaced to the high speed CM I/O bus via a VME interface board. This board allows a standard VME based UNIX host and all its peripherals to be a part of a CM system. The I/O interface computer uses UNIX IPC and other UNIX devices along with the CM specific hardware and software.

The CM graphics display consists of a CM specific board which resides in the CM backplane and a standard high resolution color monitor. The software necessary to display images runs on the front end and Sequencer. Access to this device is attained via a mechanism similar to that of *cmattach*.

Error handling facilities is distributed among all of the components of the CM system. There is communication among the components to pass error information and use of the underlying UNIX system to log errors and provide core dumps and debugging tools for user programs.

5. The Connection Machine Front End Subsystem

The control of the Connection Machine system is based in the front end processor. This section will describe the mechanism by which the front end system controls the CM data processors (see figure 2).

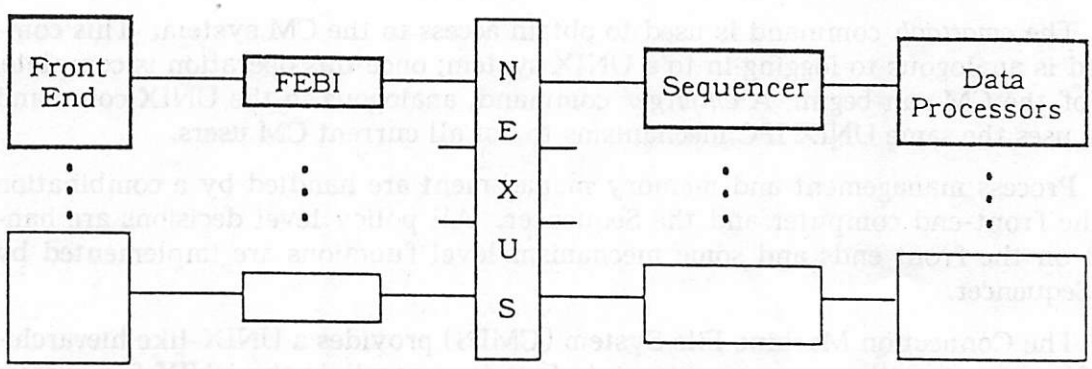


Figure 2: Front End Subsystem

The front end processor controls the front end bus interface (FEBI), the Nexus, and the Sequencer via 16 Connection Machine system registers. Four of these registers control the operation of the Nexus, while the remaining 12 access

registers on the allocated Sequencer(s). Two of these registers are used to write to and read from fifo's in the Sequencer.

The ability to rapidly deliver instructions and get results from the Connection Machine is crucial to good application performance. For this reason the Sequencer registers are mapped directly into the address space of the front end system process currently "attached" to the Connection Machine system rather than using operating system calls to access the registers.

A process accesses CM resources by first gaining exclusive access to the FEBI. Access to the FEBI is controlled by the UNIX FEBI device driver. When a process desires access to the FEBI, it first opens an "indirect" CM device. This dummy device does not initially have access to the FEBI registers, and is in fact not associated with any hardware at all, but allows the user process to call driver *ioctl()* routines.

After successfully opening the indirect device, a process can issue a CONNECT_TO_INTERFACE *ioctl()* to request the driver to allocate a real FEBI. Blocking and non-blocking versions of this call take a list of requested FEBIs as an argument. Once successfully attached to a particular hardware interface, a process can map the FEBI CM registers into its address space. This mapping is accomplished on SunOS systems via a *mmap()* entry to the FEBI device driver, and via a special *ioctl()* in the driver on ULTRIX systems.

Gaining access to the Connection Machine processors requires the further step of setting up the Nexus to connect a given front end to the desired set of Sequencers and then initializing the CM processors, memory, and Sequencers. In some circumstances, new CM microcode is loaded into the Sequencers. For historical reasons, and for compatibility across our front-ends, the code which implements these functions is currently written in Lisp.

Users running Lisp under UNIX call C stubs to obtain access to the FEBI as described above. Then the Lisp process performs the Sequencer allocation and initialization functions. For non-Lisp access to the CM we have implemented a system which actually uses Lisp to do the allocation and initialization of CM hardware. The shell command *cmattach* calls the driver to access a FEBI. If one is not available, *cmattach* can optionally wait until a FEBI is available. Once the interface is allocated, a daemon process running a Lisp subprocess is called to set up and initialize the Connection Machine system. If the daemon indicates that the requested Sequencers are not available (for example if they are in use by a different front end), the cmattach command can sleep a short time and try the operation again.

Once the system is initialized, the cmattach program forks either an interactive sub-shell from which users execute their CM programs, or executes the user's program directly. The indirect device name is passed to the user program in an environment variable. When a CM application starts up, it opens the device named in the CMDEVICE environment variable, queries the driver via an *ioctl()* to make sure that the device is still connected to the interface, and then maps the FEBI registers into the process's address space.

Error handling is also done by the Lisp system. When a CM Exception is flagged by the FEBI, the runtime system calls the Lisp daemon to probe the

hardware and interpret the error condition. Error information is written to a user definable error stream (generally *stderr*), and a user settable error function is called (by default *abort()*). Should an interface error occur, such as a parity error or bus timeout, the driver can modify the process's page table entries mapping the device to prevent further register accesses. The driver can also detach an indirect device from the hardware interface, allowing a process to be forceably detached from the CM.

Shell level commands are also provided for deallocating the CM *(cmdetach)*, finding out who is currently attached to or waiting for an interface *(cmusers)*, detaching another user or front end system from the CM *(cmdetach)*, and initializing the system *(cmcoldboot)*.

The front end system also acts as a center of control of CM I/O subsystem devices (figure 3). The simplest case is the Connection Machine graphics display system. The graphics display is connected to a interface board which resides in the Connection Machine system backplane. Under control of front end instructions, data is transferred directly from the Connection Machine processors to the display system. In order for a program to access the display system, the program must be attached to the section of the CM in which the frame buffer board resides. Thus allocation of graphics display systems is currently handled in the same manner as allocation of Sequencers and processors; the user either gets exclusive access or no access at all.

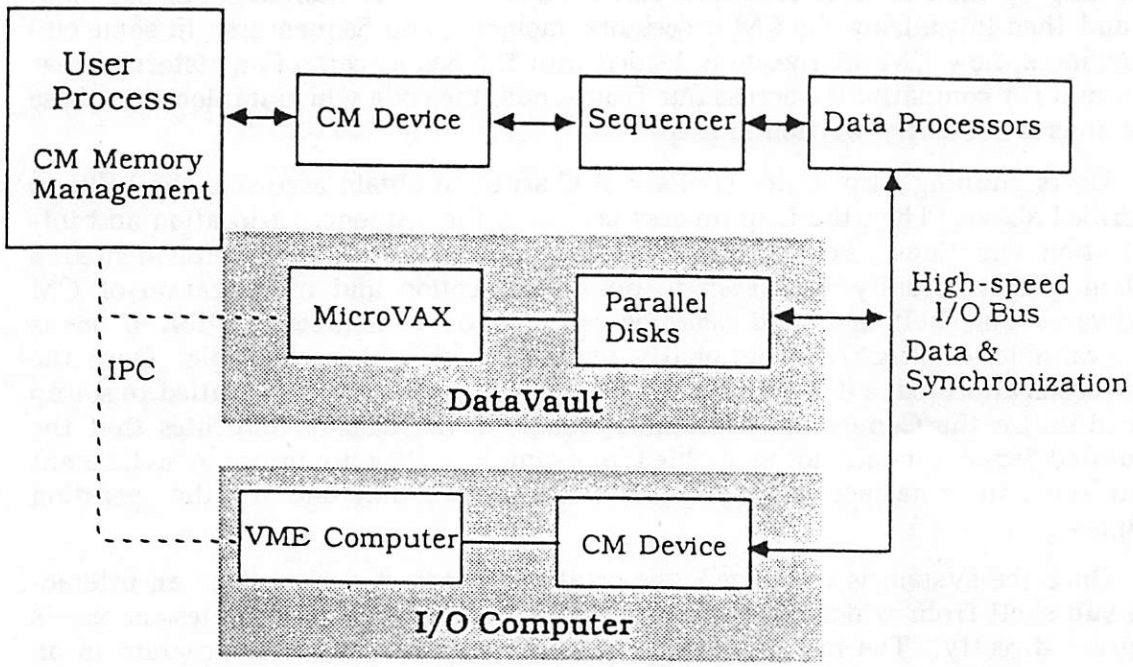


Figure 3: The front end and the I/O system.

I/O operations to the CM I/O bus are also initiated under front end control. Under front end control, the CM processors transfers data to the CM I/O controller board, and the I/O controller board is programmed to send or receive data on the CM I/O bus. Before a transfer on the bus is initiated, however, the front end system must arrange the other end of the transfer with the destination I/O subsystem controller host. These systems are are separate minicomputers running UNIX and in operations which transfer data between the CM and an I/O device, they operate in cooperation with the front end system.

When a process running on the CM needs to access data on the DataVault or retrieve data from a I/O processor, the front end system sends a command message to the I/O processor computer which then sets up the data transfer over the CM I/O bus. These command messages may be sent either via standard UNIX IPC mechanisms (TCP/IP networking), or via a command channel on the CM I/O bus. The I/O processor receives the message and passes it off to a CM I/O daemon server process. Another IPC message is sent back to the front end system to inform it of the success or failure of the operation. Data transfer is then initiated from the I/O processor to the CM via the CM I/O bus, freeing the I/O processor and the front end processor for other computations. Completion or error information is passed from the I/O processor.

6. User Process Memory Management

Memory allocation within a user process has some novel aspects that are appropriate for massively parallel machines. Data structures (memory fields) are allocated across all processors. This striping means that large arrays have elements across the entire machine. In fact, each element of a large array appears to have its own processor. Further, the dimensions of the data structures are not limited to the number of data processors because of a mechanism for simulating more processors. Currently, the CM does not support disk based virtual memory, so each user's entire memory space must fit within the physical memory of the CM (.5 GBytes for a 64k processor machine). Memory allocation within a processor is performed by the run-time system in user specified increments rather than fixed pages. These increments are often small such as a 1 bit flag or a 32 bit number in each of the data processors. This section will address only those parts of memory allocation system that are necessary for understanding the operating system. For a further explanation of the programming model see *Introduction To Data Level Parallelism.*[16]

Since the memory structure in each data processor is the same, only one map to this structure must be kept for each process using the CM. This information, called the field table, is kept in the front-end processor. Some information about the most recently used fields is cached on the Sequencer for efficiency. All allocation and field management is done on the front-end by explicit calls from the user process. In fact, the table that stores all the field locations (memory stripes and their properties) is stored in the user process's address space. This is similar to heap and stack management within a language, but this functionality has been

[16] Thinking Machines Corporation, *Introduction to Data Lavel Parallelism.* Thinking Machines Corporation Technical Report 86.14, April 1986

incorporated into the CM instruction set so that programs written in different languages can call each other. This field table is not used to share information between user processes.

The only restrictions that the current memory allocation procedure makes on the inter-user memory allocation organization is that each user must have a contiguous block of CM memory. User memory does not have to be zero based, so that a user can be located in any portion of memory. A user processes' memory is relocatable by looping through the field table, moving the data in the CM and updating the field table. The user does not have access to the physical location of the fields. Since the field table contains physical locations, relocation is expensive, but the run-time behavior is fast. Since the CM does not have hardware page tables, caching the addresses in this manner is desirable from a performance point of view.

7. Connection Machine Process Management

Currently, multiple users can access the Connection Machine system through a simple batch system. Under batch operation, a simple queue is maintained on the front-end and the data processors are completely reset between users. Each user gets sole use of the data processors, and all their associated memory and I/O devices. Under batch, only one front end process has access to the CM at a given time. Timesharing the Connection Machine between several users on the front end system can be achieved by allocating the FEBI device for short intervals to different processes as they are requested. Each process is given a fraction of the total CM memory, but when active, uses all the CM data processors. This section will describe this simple timesharing model.

Under a prototype timesharing implementation, each user is allocated a virtual CM with no inter-process communication, which is identical to the batch model of the machine with the single exception of having less memory available on each data processor (see figure 4). Timesharing of the data processors is done on top of the UNIX timesharing system of the front end. Only operations requiring the use of the CM data processors cause the system to determine whether the user can use the data processors at a given time. Manipulation of serial data, front end I/O, and user interaction all leave the CM data processors idle. Since most data parallel programs are actually a mix of serial and parallel code, most operations requiring the use of the CM data processors come in bursts. Because of this the CM process switching quantum can be quite large by traditional timesharing standards such as 1 second.

A prototype implementation of the CM timesharing system involves several pieces. Scheduling and resource management is handled by a daemon process. User processes requesting timesharing services use UNIX IPC primitives to communicate with the timesharing daemon. A structure is shared between the daemon and the user processes via UNIX shared memory facilities in order to implement an interlocking mechanism (see figure 5).

The timesharing daemon handles the registering of new users and switching between CM processes. A process gains access to the CM data processors by registering itself with the timesharing daemon. A UNIX IPC message is sent from the user process to the timesharing daemon requesting CM resources. The timesharing

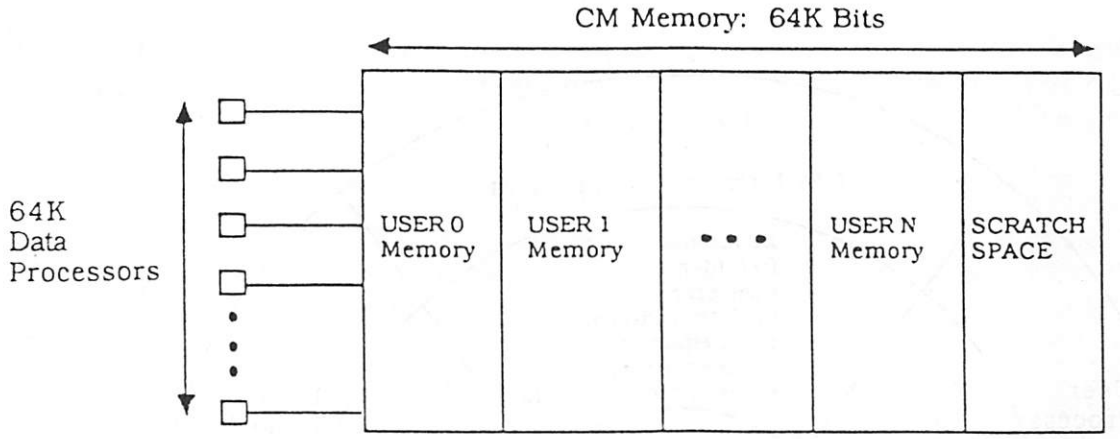


Figure 4: CM memory configuration under timesharing.

daemon sends a reply back to the user process indicating the success or failure of the request, and if successful, includes information on the amount of memory allocated to the user and a key for accessing a shared data structure. This data structure is used for most communication between the daemon and the user process.

The timesharing daemon can take the CM away from a user process anytime that process is not in a CM instruction. To implement this, all CM instructions check whether it is clear to run. If it is not clear to run then the process puts itself to sleep and waits for a signal from the daemon. If it is clear to run, then a flag is set in the shared data structure indicating that the user is actively using the CM. Once the CM instruction is completed, the flag is cleared. CM instructions are not interruptible by the daemon. To avoid a race condition, in fact, the instruction registers itself as using the CM before it checks to see if it has the right to actually use it, clearing the flag again if it is not clear to run. If the user process encounters an error on the CM it also gives up the machine, informing the daemon via another flag.

A further complication arises in handling errors that a user may have caused but not yet handled. A process may send many instructions to the CM before reading any results back. Error checking is generally done only at the time results are read back from the CM, so an error condition may exist for some time before a process notices it. In fact, one process may generate an error condition, and the CM switched to another process, before the first process is aware of the error. For this reason the timesharing daemon must check for any error conditions before switching CM processes. Since there are sophisticated error handling capabilities available to the user, an error does not lead to loss of state in the CM. The daemon uses another flag in the shared structure to indicate that an error has occurred in the background.

When more than one process is registered with the timesharing daemon, the daemon begins operation as a scheduler. Access to the CM data processors is granted in a simple round robin mechanism, with the daemon granting a process

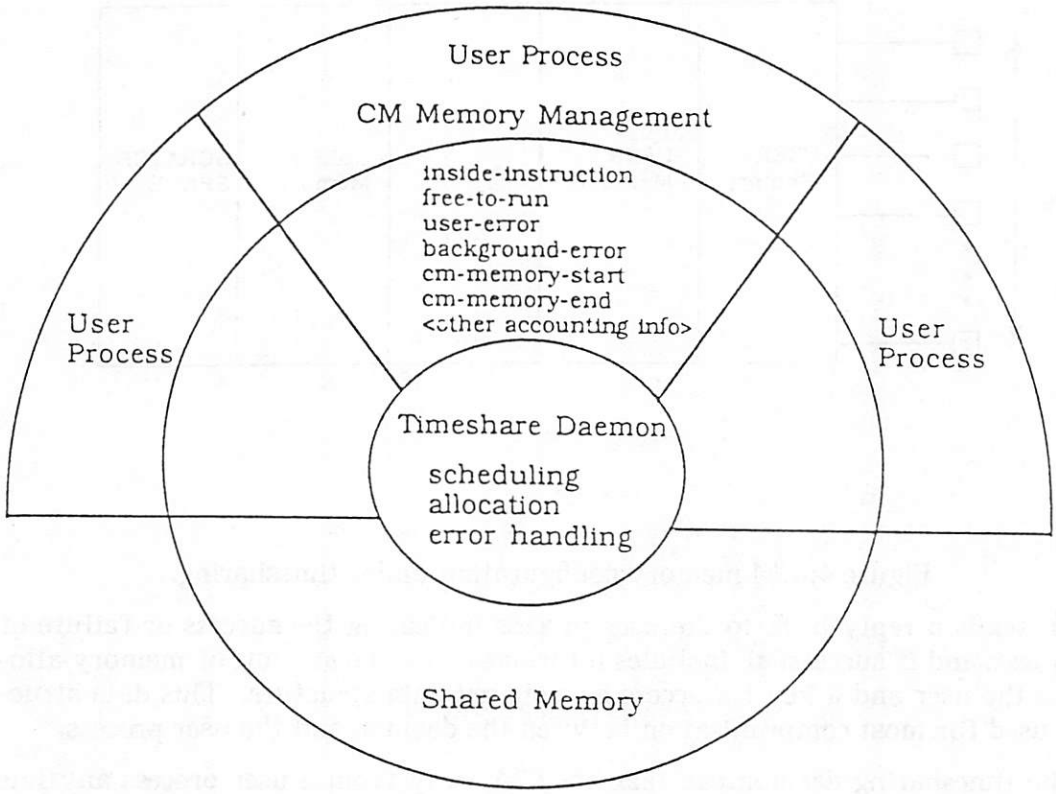


Figure 5: Timesharing control mechanism.

access to the CM by setting the appropriate flag bit in the shared data structure, and sending the user process a signal if it is waiting for CM resources.

Inter-user protection is accomplished by a variety of mechanisms. User state in the sequencer is unloaded and loaded again by the timesharing daemon process. User state in the CM memory is protected by hardware bounds registers that keep users from altering each others memory (either accidently or otherwise). I/O operations are left to complete before processes are switched, thus achieving safety and simplicity while sacrificing some performance.

As in all operating system code, code ordering is important to prevent race conditions. Since the master is not in the kernel at this point, many of these considerations are exacerbated. CM instructions are simple writes to the FEBI registers, so it is extremely unlikely that the UNIX system will block the current CM process for an extended period of time. Timesharing on a massively parallel machine needs further design and implementation work.

8. The Connection Machine File System (CMFS)

The Connection Machine File System is a high speed, parallel, hierarchical file system. It runs on a parallel disk subsystem called the DataVault which stripes bits across 32 data carrying drives and places error correcting code information on 7 additional drives. The software and hardware is designed to maximize throughput for large data transfers. The programming interface is very similar to the UNIX file system with the read and write primitives causing data to be transferred from each virtual processor in the CM.

The CMFS has two main components. They are the file system library which is linked in with the user program and runs on the CM front end processor, and the CMFS file server processor which runs on the file server computer inside the DataVault. The system operates under a client/server model where state is maintained between calls to the file server. A simple file system protocol runs on top of UNIX IPC which provides the communication path between the clients and server which run on separate computers.

The file server provides pathname resolution, logical file to physical disk block mapping and block allocation. Although there can be more than one CM front end using a DataVault, the distributed file systems issues are substantially reduced by having a single server responsible for the file system. The division of labor between the client and server in the CMFS is similar to that in NFS[17]. The CMFS, however, uses a separate namespace for file names and uses a separate set of calls to access CMFS files. The separation was necessary to allow for large data transfers to occur in single I/O operations directly from the high speed CM I/O bus. The CMFS implements an extent based data layout. A file is made up of a variable number of extents where each extent is a variable sized, physically contiguous set of blocks on the disks. Only data is stored on the set of parallel disks while all directory and inode type information is stored in the UNIX file system on the file server computer.

9. General Purpose I/O Computer

The I/O Computer can be any VME bus based computer running UNIX. A VME interface board is used to transfer data from this system to the proprietary CM I/O bus. This interface (currently under development) allows standard computers and their peripherals to transfer data to and from the CM system at high data rates. By interfacing to a standard VME a wealth of peripherals available for those systems is made available to the Connection Machine system.

The software interface between the I/O processor and the CM system is the same as that used by the CM File System. When moving data to and from the DataVault the I/O interface computer acts like the client and executes almost the same code as the CM front end computer executes when using the DataVault. The I/O Interface Computer can also act as the server when exchanging data with the CM. In both cases UNIX IPC and other UNIX devices are used for the operations. Since the I/O bus speed of the CM is 64MBytes/sec, this interface is limited only by the speed of the VME bus and the peripherals involved in the transfer.

---

[17] NFS is a trademark of Sun Microsystems, Inc.

10. Future Work on Massively Parallel Operating Systems

Operating systems for massively parallel computers, like the Connection Machine, open many design issues. Many of the performance characteristics for paging and user loads, for instance, are different for these systems. Thus the opportunities for significant performance and functionality increases are unfolding. Some of the areas that need further study are:

[1] Sophisticated Timesharing

[2] Virtual memory: Paging and Swapping

[3] Integrated I/O subsystems

[4] Remote Access

Timesharing systems could, for instance, be extended to have a process model that supported IPC, forking, and shared memory, but the specifics on how this would perform or even be designed for such systems is not understood. Further, asynchronous I/O might boost system throughput of a timeshared system. Another issue is the desirability of tightly coupling the timesharing system with the front-end kernel to make smaller Connection Machine time quantums efficient. An efficiently timeshared parallel computer would greatly extend the number of potential users and make very large systems more affordable. Accounting and performance analysis on a machine like the Connection Machine is an interesting problem because of the number of independent processors working on one program.

Paging and swapping could extend the data sizes users can easily compute on from .5 GigaBytes to 10-100 GigaBytes. How the page tables should be managed needs further study of existing programs based on how memory access patterns of data parallel algorithms. Caching performance is also effected in interesting ways in the common data parallel applications.

Various input/output devices can be used to make data-parallel machines more useful. One can imagine the uses for high speed general purpose networks, frame grabbers, music synthesizers, speaker systems, FAX machines, and the like, if they were cleanly interfaced and controlled by a large compute engine. Keeping integrated control over a large number of different high speed devices, some of which might be physically distant, is an interesting task of distributed control.

Using a high speed computers as a compute server in workstation environments opens interesting issues of data transfer, simulation, and splitting of user programs. Many institutions will have large parallel computers accessible on networks of various performances (from 100MBytes to 56k Bits). How users can best use these resources is not understood. Handling this smoothly will put demands on networks and operating system code.

Many of the system performance characteristics are different enough in a massively parallel computer, that the many operating system issues need to be re-examined. We look forward to research in this area of operating system design to increase the usefulness and performance of massively parallel computer systems.

11. Conclusion

UNIX is an important vehicle for unifying the distributed parts of the Connection Machine system. Instead of implementing a native UNIX for the CM, we have used serial machine's implementations on different platforms and have built what we need on top of it. Massively parallel computers share many of the system demands that a serial machines do, so we can use the interfaces and philosophy of UNIX in implementing operating system components for the Connection Machine system.

Further work needs to be done to make the Connection Machine operating system a true multi-user, virtual memory system. Since the control issues and memory use statistics are different from serial machines, some of the current OS work can be exploited directly while other aspects need to be examined anew.

PARX: A UNIX®-like Operating System for Transputer®-based Parallel Supercomputers

Y. Langué and T. Muntean

University of Grenoble
IMAG-Laboratoire de Génie Informatique
BP. 53X, 38041 Grenoble, France

ABSTRACT

Our paper presents the structure of a UNIX-like operating system for transputers based supercomputers. UNIX user virtual machine is a pipelined multiprocessor machine. We develop this view and introduce a parallel programming model for closely coupled multiprocessors running UNIX. Our target architecture is a network of transputers. Inmos transputers[INM85] are a new range of processing devices dedicated to build multiprocessor machines with no shared memory. They are based on a CSP[HOA78] like model of programming and their machine level language is a high level parallel programming language called occam[INM84]. Occam is more than a programming language, it is also a formalism to design parallel systems correctly. We introduce some occam constructs into the UNIX programming model.

1. Multiprocessor Architectures

1.1. Classical Approaches

Classical attemps to exploit multiprocessor architectures have hidden physical parallelism from user level. When many central processing units were running in parallel, the only visible consequence for user applications was a better response time and servicing.

Most multiprocessor computers achieve hardware parallelism. Many processing units, I/O controllers, memory managements units or coprocessors run in parallel or in quasi parallelism. However, only the lowest part of the operating system needs to take into account physical parallelism. Isolating hardware related features is often suitable because it requires less adaptation effort from existing materials and allows to implement the same software on different computers. However, operating systems cannot fully benefit from underlying parallelism and this limits noticeably computers performances regarding parallelism.

Multiprogrammed operating systems are parallel programs which take advantage of physical devices (interrupts and exceptions) to run in parallel. They emulate a certain number of virtual machines, one for each user. Unfortunately, few operating systems offer parallel or concurrent virtual machines to application

This research is supported by the Esprit Project P1085 of the EEC.

programs. Some operating systems, such as UNIX, offer a form of concurrency and communication primitives between virtual machines.

We are mainly interested in user level parallelism. One of UNIX most successful feature is the ability to create a clone of an executing process, thus providing both a form of concurrency and communication channels between processes. UNIX *pipes*, however powerful, does not allow to fully take advantage of this concurrency: they enforce a communication protocol that is not suitable for fine grain communication oriented processes.

Multiprocessor networks allow to achieve fine grain parallelism and communication, provide high level virtual machines are maped closely enough to physical machines. It is the purpose of the Supernode project.

1.2. The Supernode Multiprocessor Architecture

We introduce the Supernode, a transputer based reconfigurable parallel architecture of supercomputers which can be used in both MIMD and SIMD modes (SPMD to be precise).

To help providing fine grain parallelism, the transputer hardware manages a limited form of process abstraction. Moreover, processors physical interconnection network can be changed dynamically. These two features can be used to provide a refined maping between physical machine and user conceptual machines and to reduce the cost associated with interprocess communication. Light processes within a virtual machine can be directly maped into hardware processes. As target application programs include such fine grain parallelism that instructions come to be consider as to be processes, interprocess communication grain should be tuned accordingly. The limited number of external communication links is a bottleneck to remote processes communication. Reconfigurability allows physical links to be adjusted when needed. The Supernode can then be adapted to any topological problem configuration. Dynamically adjusting network topologies envolves a cost that could be prohibitive compared to introducing routing mechanisms. We are investigating how to tune communication and light process grain for efficient use of reconfigurability.

The reconfigurability of the Supernode allows to obtain any possible topological configuration of the machine while executing parallel programs. Dynamical configuration is achieved with a programmable switch device which allows modular construction of higher level Supernode machines and ensures proper interprocessor communication within any configuration. Three keywords are at the basis of the Supernode achitectural model: modularity, homogeneity and extendability. A Supernode is a collection of lower level machines which can be Supernodes or transputers at the lowest level. Within a n-level Supernode, each (n-1)-level Supernode has an amount of physical links dedicated to external communication. A n-level switch is responsible for establishing link to link correspondances between (n-1)-level links. It is capable of performing any possible link to link interconnection. The switch is dynamically reconfigurable under the control of a control processor attached to each level. Each module has the same regular structure: a set of sub-modules interconnected via a switch. Extending the machine is just a matter of adding another level of switching. The basic module is a collection of working processors each having its own local memory (from 256ko

of fast access memory to 4Mo of DRAM per transputer), interconnected through a switch. The switch also allows server processors to be added at each level in order to provide system services such as memory, disks servers or I/O dedicated controllers. The basic module is controlled by a processor that masters a control bus where working processors are connected.

1.3. The Transputer

Each transputer is based on a 32 or 16 bits processing unit that includes link communication devices, a floating point unit and a private memory extending from (today) 2kbytes fast internal memory on chip to 4 Gbytes of external memory. Four serial bi-directional links are available for external communications. These links can be directly connected to other transputer links or through link interface adaptors to other hardware devices. The link interface implies that transputers networks are not classical local area networks but genuine multiprocessor networks. There is no shared memory, information exchange is explicitely done using messages sent through links. Links throughput is 20 Mbits/s.

The transputer has hardware support for concurrent execution. It implements and manages a process abstraction. Each process has a process descriptor which records process status and hardware resources usage (timers). The transputer improves context switching by doing microcoded process scheduling. The context of a transputer process is a few words (five 32-bits words on a T800, which brings process context switch to 600 ns). There are two levels of priority. Both priorities process scheduling is communication driven. Furthermore, low priority processes are time-sliced while high priority processes are not.

UNIX model of processes does not correspond to that of the transputer and even the recent developments in distributed operating systems based on UNIX do not feature microcoded process scheduling.

2. UNIX in a Multitransputer Environment

UNIX is one of the few operating systems that offer a form of concurrency at user level. Interprocess communication primitives provided correspond to a large grain of parallelism. We describe how UNIX principles can be applied and implemented on a closely coupled multiprocessor and especially on Supernodes. Appropriate parallel programming model and paradigm are proposed at user level.

2.1. Processes

The basic executing entity under UNIX is a process. The process abstraction encompasses I/O, memory and resource management. The execution model is a sequential process running on an abstract machine. A program is executed as a single process, and the system endeavours in provinding it with as much processing power as possible. When one aims at concurrent programming, it migth be painful to use such an execution model. A lot of work has been done on UNIX, in particular to refine the executing entity grain[Apo87, GAR87, TAN81]. Light weight processes, tasks or threads have been introduced within the sequential process model. Introduction of light weight processes within a sequential process model can be obtained by modifying run-time environment or re-designing operating system kernel.

The first approach is the most commonly found because it requires relatively little re-design effort. The drawbacks are immediate; because the light weight process environment is built upon a heavy powerful sequential process model, some basical concurrent constructs clash with the underlying concepts. A great part of the virtual machine power is lost for light weight processes, especially blocking system calls.

Re-designing the operating system kernel requires a great deal of job. Hardware concepts evolution which tend to match operating system programming models contribute to integrate efficiently task management control structures. Unfortunately backwards compatibility sometimes refrains innovation.

An original approach is to let the hardware manage light weight processes, while the operating system builds higher level parallel program abstraction upon the basic process concept. This is the direction we investigate.

For interprocess communication the sequential process is kept as the execution mode in UNIX. To perform communication, a third party is required, as processes don't interact out of operating system supervision. Depending on the communication protocol, third parties can be sockets addresses, mailboxes or other system-owned objects managed by the file system.

Establishing communications between processes in a parallel environment should be thought of as establishing an interconnection network between processes. As communication is heavily used in these environments, it consumes a significant fraction of execution time and communication overhead must be reduced[DUD88]. Multiprocessor networks without shared memory require message passing as basic primitives for interprocessor communication and synchronisation. A fundamental issue is to determine the grain of parallelism and communication that could be implemented on a multiprocessor machine.

Monoprocessor memory management policies are most often tied to the existence of multiple levels of memory: primary memory, secondary memory and so on. The different storage levels have very different access times. One of the memory management goals is to provide a large and fast access virtual memory space to processes. A multiprocessor computer does not have swap devices handy for each processor. Information must often be passed from processor to processor before reaching the final receiver. It is therefore clear that memory management must be revisited. System calls designed in the basis of common physical memory underlying virtual memories have to be revisited. For instance, the sole means of process creation, the *fork()* system call, finds most of his efficiency in the fact that parent and child can share code and great portions of data, without lost of security. On a multiprocessor system, this is not the cheapest way to create a new process core image.

It has been observed that in most cases, processes using large memory space do not access to memory uniformly. When memory is accessed at random, as it has no hardware memory management unit, the transputer cannot benefit from this observation. On the contrary, when memory is accessed via procedure calls, such as in read or write calls, software memory management algorithms can be implemented, mainly buffer cache techniques. In closely coupled processors environment, we can assume that processing power is tightly related to memory space.

Then allocating memory space is the same as allocating computing power and hence processors with more or less local memory.

2.2. File System

The file system has a primary function under UNIX. Resource servers have been maped either in the virtual machine or in the file system. It is the sole resource server. Resources that can be shared are maped in the file abstraction. UNIX file system is used as a handy resource both for administrative and user-application purposes. The file abstraction of devices and memory located objects has been developped under UNIX, making life easier to application programmers. This file abstraction allows to use the same operations for accessing resources as different as files, pipes or sockets. Resources are essentialy passive, operations on active entities such as processes are not enclosed within the file abstraction.

With light processes must be added corresponding communication primitives for parallelism. Classical UNIX communication protocols have to be enriched. The file system should be able to bind names to communication channels as it does for sockets, thus providing the additional name server function for the network.

Other file system functions such as providing swap devices or accessing the whole addressing domain on a monoprocessor are not easy to cope with. We have already discussed memory management. Due to memory distribution over processors, maping physical memory into the special file abstraction is expensive. This particular usage is limited to administrative purposes and can be neglected since equivalent effects can be obtained by other means.

In distributed systems based upon common network topologies (bus, star, ring etc.), viewing files as a cheap resource is successful, because the communication medium is available to each processor. Each processor, having direct access to the network, can exchange information with any other one at relatively low cost. A multiprocessor environment does not offer such facilities. When fine grain parallelism is challenged, the file system is bound to be distributed or split up into thiner file systems.

2.3. UNIX Programming Model

UNIX processes heavily use computing power and communicate rather rarely. Creating a new process is an expensive operation leading to a clone of the originator. Whenever processes communicate, communication is lengthy and counterparts are not likely to quickly switch to other correspondents. Communications and access to system resources is interfaced through the file system.

Process creation is performed with the *fork()* system call which duplicates the originator process. UNIX basic construct to build a network of processes is a double feet fork. Only one process is created at a time and a parallel execution graph under UNIX is a binary tree. The double feet fork allows to construct any process network but is a very primitive means of process creation for fine grain parallelism. Process creation is often followed by a call to replace executing core image. This scheme is the only way to obtain concurrent process execution and its heavy usage demonstrates a need to be provided with much more concurrency.

Interprocess communication is performed through shared channels descriptors with two extremities (pipes, sockets, mailboxes). Two processes can use a communication channel even if no common ancestor ever created the channel. This possibility should be extended to pipes. Naming pipes would allow to easily write dynamical servers.

Concurrency at user level in UNIX is achieved only through process duplication and shared channels. UNIX *shell* program make intensive use of concurrency. It constructs pipelined processes when communication is required and a process tree when it is not. This distinction corresponds to communication and computational views of the same UNIX abstract machine. This point immediately brings in mind the idea that UNIX should fit very well on a multiprocessor machine, it would be sufficient to allocate a physical (or virtual to some degree) processor to each member of a computational tree or pipeline. We are investigating these directions through the use of Minix, a UNIX-like operating system proposed by Tanenbaum[TAN87]

3. Minix on a Supernode: PARX

Minix, a simple UNIX-like operating system, is presented with its implementation on a Supernode. Minix structures the system into processes of three categories: user processes, system servers and tasks. We use Minix as a UNIX-compatibility test for the Supernode.

Minix is divided into four layers that we can illustrate in the following picture:

4. Init and user processes
3. Server processes (memory management and file system)
2. Tasks, one for each device class: disk, tty, clock, system
1. Process management and message passing

One of Minix attempts is to break the UNIX monolithical structure. To that purpose, the process abstraction is implemented in the most basic layer, while device drivers, memory manager and file system are simply processes. Level 1 is the abstract machine necessary to implement process abstraction and the basic interprocess communication mechanism: message passing. Interrupts and traps are transformed into messages for upper layers. This layer is called Minix kernel. A task is a special high priority level process driving a device. It uses kernel functions to achieve hardware dependent operations such as reading or writing at specific memory locations. Let us notice that clock and system services run as tasks, and therefore processes. Server processes are high priority user processes identified by the kernel. They exchange messages with system tasks to obtain system services. User processes send messages to server processes which are their sole relation with the system. The calling sequence from user processes to tasks guarantees that the system never deadlocks. User processes block when calling server processes. Server processes request operations from tasks without blocking. Tasks reply at agreed memory locations. The different layers are functionaly independent. From task layer to upper layers, Minix can be viewed as a collection of servers.

- The Kernel (layer 1)

Two primary kernel functions are process scheduling and message passing. A communication sub-layer has been defined, which passes messages from ports to ports. A port is a memory object used to send and receive messages in a synchronous or asynchronous way. A port identifies a recipient address on a processor. Weaker synchronisation schemes can be built above this mechanism. A collection of five basic protocols are provided for upper layers interface:

— point to point rendez-vous (CSP like protocol),

— client(s) to server (blocking send and receive),

— server to task (blocking send with a recipient port for reply),

— read, write or execute at specific locations on a remote processor (direct memory access and remote process activation),

— a basic diffusion protocol.

These protocols were identified as to fit not only Minix own requests but also a large variety of other communication schemes for application processes. They were designed in such a way that they can be freely mixed and interleaved. The idea is that a specific protocol is attached to each communication object. This sub-layer is implemented on each transputer but any subset can be implemented. Communication driven process scheduling as well as time-slicing for low priority processes is done by the hardware. The user has no complete control over this scheduling. Instead, one can control most of the hardware scheduling queues. Since there is absolutely no protection on the transputer, we have enlarged the protection domain to be the processor itself. Therefore, a Minix process runs on one or several transputers, possibly using many transputer processes. Furthermore, a Minix process is always scheduled and must never compete for the processor. Kernel process tables are managed on a special processor, we call it a system processor. The Supernode architecture provides such a processor at each level of the machine. User and supervisor modes are achieved by running the kernel on a different processor from user processors.

- Tasks Layer (layer 2)

Drivers are independent and can be attached to the devices. If several geographically distant resources must be viewed through a single object manager, a task server process is made responsible for that abstraction. Tasks manage drivers and handle requests for reading or writing. As most of them are asynchronous, they send replies to their clients and no rendez-vous is ever established.

- Servers Layer (layer 3)

Two primary servers co-exist in Minix: Memory manager and File system. Memory manager is responsible for allocating and releasing memory and interfaces system calls involving memory management. These are fondamental to UNIX, such as *fork()*, *exec()*, *signal()* etc. Memory manager must provide memory protection between processes. The common and most efficient way to do this is to require help from the hardware. Since no memory management unit is available for the transputer, PARX keeps a single process on each processor or processors cluster.

With adequate underlying communication protection, memory security is achieved. Memory allocation then turns out to be processor allocation.

File management is performed on disk control and file system processors. Little changes are needed to the file system. The great problem is to transfer information at acceptable rate. This involves maintaining caches on file system behalf at several processors on the path from a file system to it clients.

Implementing PARX helped us to establish significant results. To reach the grain of parallelism that is challenged, light processes must be managed as close as possible from hardware processes. Programs must be given the true machine parallel facilities, thus being responsible for process management. We are investigating several suitable distribution schemes for accessing resources. Currently, reconfiguration cost does not allow fine grain access to resources. Allocating processor clusters to programs rise problems of protection and resource coherency. The control overhead should be minimal in order to maintain a fine grain of communication.

4. The Parallel Program Model

Languages induce or are semantically based on programming models. Most classical languages do satisfy themselves with a sequential programming model. This is not true for occam, the machine language of the transputer, as for most concurrent languages. From the programming models one wants to offer, a set of operators or constructors allowing to derive a formal and proovable process model must be defined. Machine administration policies induce other constraints. Resource management policies and user processes management lead to the design of a user abstract machine model. The actual machine must be able to correctly emulate a certain amount of such user abstract machines.

CSP derived programming models demonstrate to be a good paradigm for parallel programming. UNIX and occam programming models serve as general guidelines towards an original parallel programming model which is not within the scope of this paper, but some ideas are presented hereafter. Resources allocated are virtual processors which could extend from a fraction of a physical processor to a collection of processors. Two goals are challenged: efficiency in execution and availability for interactive users.

In the occam model, which is derived from CSP, a sequential process is still the executing entity but is thought of essentially as a communicating entity. Processes use little computing power but do a lot of process creation and communication. The transputer was designed to closely fit the occam model of programming.

Occam operator to create a network of processes is the *PAR* construct. The basical process construct is an occam lattice[1]. In the occam programming model, two processes can use a channel to communicate if and only if the channel was created by one of their common ancestors.

[1] An occam lattice is a complete lattice (L, <) where Top and Bottom elements (1, 0) are unique and elements different from 1, 0 are incomparable.

We define a stage of a parallel program as to be the set of its component processes and their interconnection network at a given moment of program execution. We compare programs using the measure of the minimum number of transitions a program must execute from a given stage to reach a given target stage. We must insist on that establishing a communication channel counts for a transition as this action modifies the interconnection network. At first sight, UNIX and occam programming models are equivalent regarding communication transitions. UNIX has two advantages over occam: it offers different communication protocols and it is not compulsory that communication channels be created by grand-father processes. As regarding process creation transitions, occam has an advantage.

We now introduce the parallel program model, which unify previous models advantages and corrects their pitfalls.

To achieve fine grain parallelism on a multiprocessor computer, a program must be managed from two complementary views: internal parallel processes and global program behaviour in its environment. The first view corresponds to run-time virtual machine management while the second is an operating system specific job.

- Parallel Program Run-time Virtual Machine

The run-time virtual machine can construct a specific process model upon the basic hardware process. As we are concerned with communicating sequential processes, we define a parallel program as a set of couples (p_i, g_i), where p_i is the set of processes at stage i, g_i the execution graph at stage i. A transition is defined as a change in the communication network or a process creation or termination in the execution graph. p_i and g_i are derived automatically from program analysis [MUN88]. The run-time virtual machine is responsible for the allocation of processors and communication links necessary to execute the program. A maping is then established between (p_i, g_i) and the physical network resources.

At a given stage, an executing program could form a "closed" network, while at others it communicates with another executing program, thus requiring operating system services. An executing program dynamically adapts itself to its job. Light processes are maped closely to hardware, so that they fully benefit from fine grain parallelism and communication primitives. High level communication objects can be constructed by the PARX communication kernel in order to require external services.

- Parallel Programs Operating System

Each parallel program require a cluster of processors. Each stage in the program execution must be provided with the corresponding processor network. The parallel program model in PARX is as an automaton which allocates clusters of processors to programs. It manages processor, schedules programs and provides a network system environment. This feature is achieved by changing the physical network topology staticaly, quasi-synchronously or synchronously using the Supernode reconfigurable switch.

A parallel program can be viewed as a collection of communicating sequential processes. From an operating system point of view, an executing program is then a series of couples (set of processors, interconnection network). Parallel programs also need to communicate with their environment. For instance a read request to

the file system requires a communication channel to be established between the client and the server. As parallel programs execute asynchronously, static synchronisation cannot be achieved as within a single parallel program. Such communications and synchronization must be established dynamically. A communication kernel is responsible for the communication.

- The PARX Communication Kernel

Resources and services are required by parallel programs and not by light processes within a program. Light processes are not visible from outside the program. A channel allowing to request resources and services is predefined for each parallel program. In response to such requests, the communication kernel opens communications with other programs so that parallel programs environment acts like another parallel program. This is done by maping a channel (virtual or physical) from a client parallel program to a counterpart server interface channel. Global servers maintain free channels to their clients disposal. The communication kernel retrieves correspondants which can be located anywhere in the machine.

Programs require services from servers. When the communication kernel opens a channel between a client program and a server, client requests must obey to server's protocol (to channel protocol). For instance, writing on a pipe can be a blocking operation whereas writing ordinary files is not.

Having defined a program as a set of couples (set of processors, network interconnection) or (set of processes, execution graph), creating a program or a process requires a new interconnection network or communication graph. We have seen that these two views correspond to operating system and run-time environment views of a parallel program. Compilers and run-time libraries deal with process creation and management and create communication channels between processes while operating system manages parallel programs, providing processors and physical network topologies.

PARX programming model design is relatively unusual but demonstrates to be both efficient and meaningful to programmers. It express both physical parallelism and concurrency, and allows to manipulate processes at free will. Furthermore, different run-time environments can be provided in homogeneous way, for instance occam, UNIX, parallel Prolog among others.

As concurrent processes are allowed within a single program, process creation includes caller duplication and loading new process core image. The model allows all UNIX system calls, although with different underlying implementations and costs, and offers additionnal features borrowed from the CSP programming model such as light process alternatives or guards for non deterministic control of execution.

Conclusion

This paper focuses on the operating system aspects of an original family of supercomputer architectures. A UNIX approach is developped to define operating system requirements. We exhibit parallelism and discuss the grain that should be provided with it. An original parallel programming model is derived that combines advantages from the Supernode architecture, UNIX and occam programming models.

PARX parallel program model allows programmers to express the communication network they intend to establish between parallel processes and additional information such as the kind of synchronisation they want in data exchanges. Different levels of services can be provided, depending on the run-time virtual machine a program chose to run.

Acknowledgments

Special thanks are due to Jacques Briat for his help, advice and assistance.

References

[INM85] INMOSINMOS, *TRANSPUTER*, 72-TRN-006-001, September 1985. Reference manual

[HOA78] HOARE C.A.R, HOARE, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, August 1978.

[INM84] INMOS Limited, INMOS, *OCCAM programming manual*, Prentice-hall international, 1984.

[Apo87] Apollo Computer Inc., APOLLO COMPUTER, "Concurrent Programming Support (CPS) Reference," *Software Release*, 1987.

[GAR87] Perihelion Software Ltd, N. H. GARNETT, "Helios: An Operating System for the Transputer," *Programming of Transputer based Machines*, Grenoble, 14, 16 Sept. 1987.

[TAN81] TANENBAUM A.S, MULLENDER S.J, TANENBAUM, "An overview of the AMOEBA distributed operating system," *ACM Operating Systems Reviews*, vol. 15, no. 3, July 1981.

[DUD88] IMAG-LGI, University of Grenoble, A. DUDA, "On the Tradoff between Parallelism and Communication," *4 th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Palma de Mallorca*, Palma de Mallorca, September 15-17 1988.

[TAN87] Vrije Universiteit, A. S. TANENBAUM, *OPERATING SYSTEMS, Design and Implementation*, , Amsterdam, The Netherlands, 1987.

[MUN88] T. MUNTEAN, "Parallel Programming of Transputer based Machines," in *7th Occam User Group*, ed. T. MUNTEAN, Amsterdam, 1988.

[BAC] M. J. BACH, *The Design of the UNIX Operating System*, .

[JOF81] Computer Systems Research Group, Berkeley, W. JOY & R. FABRY, "An Architecture for interprocess communication in UNIX," *Draft*, Berkeley, University of California, June 22 1981.

[BCM86] Computer Science Department, Wisconsin, Madison, M. BHATTACHARYYA, D. COHRS & B. MILLER, "Implementation of a visual UNIX process connector," *Tech. Rep. 677*, Madison, Wisconsin, December 1986.

[SHI87] Centre for Mathematics and Computer Science, Amsterdam, I. SHIZGAL, "An Amoeba Replicated Service Organisation," *Reports, Centre for Mathematics and Computer Science*, vol. CS-R8723, Amsterdam, The Netherlands, 1987.

[JLH88] University of Washington, E. JULY, H. LEVY, N. HUTCHINSON & A. BLACK, "Fine-Grained mobility in the Emerald System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 109-133, Febuary 1988.

[RAS81] RASHID R.F, RASHID, "ACCENT: a communication oriented network operating system kernel," in , Proc. of the 8th ACM Symp. on Operating System Principles, 1981.

[ZAY87] Carnegie Mellon University, E. R. ZAYAS, *The Use of Copy-On-Reference in a Process Migration System*, , 1987.

[NG87] Southampton University, E. NG & G. S. PANESAR, "Switch Object," *Working Paper Esprit Project P1085*, vol. 53, 28 September 1987.

[JEN86] Southampton University, C. JESSHOPE & D. NICOLE, "Supernode Architecture Evaluation," *Report Esprit Project P1085*, vol. 003, 22 May 1986.

MULTITASKING UNDER UNICOS: EXPERIENCES WITH THE CRAY 2

MARTIN FOUTS
MATHEMATICIAN
NAS SYSTEM DEVELOPMENT BRANCH
NASA AMES RESEARCH CENTER
FOUTS@ORVILLE.NAS.NASA.GOV

ABSTRACT

A two dimensional elliptical partial differential equation solver was coded in Fortran using multitasking and microtasking methods under UniCos on the Cray 2. Analysis of the performance of this solver led to the discovery of two difficulties in achieving speedup in a multiuser environment. Investigation of poor speedup in multiuser mode led to an understanding of the limitations of task scheduling under UniCos.

A prerelease of microtasking under UniCos 4.0 was evaluated. Microtasking proves to be easier to use and does not suffer from the same overhead problems as multitasking, but does still have some unexplained performance issues.

This paper describes the implementation of the solver, discusses experiences gained while debugging the program, describes the limitations found, describes a model of process scheduling used in the analysis, and proposes solutions to some of the problems.

1. Introduction. Cray Research Incorporated (CRI) has implemented a multitasking support library under UniCos on the Cray 2. Researchers at Ames have begun to experiment with this facility with disappointing results. This study was undertaken in an attempt to identify the problem areas. After the study was undertaken, a CRI facility called microtasking was made available, on the Cray 2, and was included in the study.

In order to understand multitasking in an environment similar to that in which it is used by researchers, a program was written which implements a finite difference method solution of a two dimensional elliptical partial differential equation (ellipse). The problem was chosen because it is similar in computational requirements to problems solved by users, but is easy to implement, and has an analytic solution, which can be used to verify that the implementation is correct.

To concentrate on the multitasking issues, a simple program was used. A five point Seidel solver for the two dimensional Helmholtz equation with constant boundary values was obtained from [1]. This program was first validated against the test case from the book, and then vectorized.

Once the program was completely vectorized, the Seidel solver was multitasked using two different ways. The problem size was increased by increasing the size of the mesh and the number of iterations of the solver. Results from each of the three programs were compared to ensure that the translations were accurate and timing data was gathered.

After this had been done, UniCos 4.0 became available on the Cray 2, including a first implementation of the microtasking facility. The solver was reimplemented using

microtasking, and timing data was gathered.

The problem being solved is easy to parallelize, since the method used can be divided into independent subtasks in several different ways. Since the problem can be partitioned to fine granularity, it should be possible to achieve speedups approaching the number of processors used. Neither of the multitasking methods were able to achieve such speedup either in standalone or multiuser timing. The microtasked implementation achieved a much higher speedup.

UniCos multitasking and microtasking are described. The ellipse program is described, findings are presented, and ways of dealing with the limitations are presented.

2. UniCos Multitasking Background. A complete discussion of the UniCos Multitasking library can be found in [2]. This discussion is only intended to supply background.

To implement a shared memory multitasking facility on the Cray 2, CRI introduced a new system call `tfork`, which behaves like `fork`, except that the resulting new process is in the same address space as the parent process. In the Cray literature, sibling processes created by `tfork` are referred to as *Logical CPUs*.

The Fortran programmer does not invoke `tfork` directly, but rather calls the subroutine `TSKSTART` which creates a *task*. A multitasked application is one UniCos process consisting of one or more `tfork` generated sibling processes (logical CPUs) which are used to execute one or more tasks. There does not have to be a one to one correspondence between logical CPUs and tasks, or between logical CPUs and physical processors. The usual case is to create a logical CPU (process) for each physical CPU in the system, and to create a number of tasks which corresponds to the partitioning of the problem. Frequently, the partitioning calls for more tasks than logical CPUs.

When the number of tasks is larger than the number of logical CPUs, the *library scheduler* is used to schedule tasks. This level of scheduling is nonpreemptive. When a task has been assigned to a logical CPU, that task runs until it explicitly gives up the CPU by calling one of the blocking routines described below.

This design, involving an application level nonpreemptive scheduler as well as the kernel level process scheduler was implemented to avoid the high cost of process level context switching. This tradeoffs is only advantageous if the introduced overhead due to the library scheduler does not exceed that saved from reducing the number of context switches.

The cost of starting a process under UniCos is relatively high. Mean time to execute a `fork` is on the order of 250 milliseconds on the Cray 2, while time to execute a `tfork` is on the order of 500 milliseconds. This time has been seen to vary from under one millisecond to over one second, and appears to be heavily dependent on system load and other factors outside of the programmer's control. Because of this overhead, the usual mechanism for multitasking is to execute `tfork` several times early in the application program, and to keep all of these sibling processes active until the program is terminated.

In addition to starting a task with `TSKSTART` and waiting for it to complete with `TSKWAIT`, the multitasking library provides:

- critical region locking, (LOCKASGN, LOCKON, LOCKOFF, and LOCKREL),
- event based synchronization, (EVASGN, EVWAIT, EVPOST, EVCLEAR, and EVREL),
- barrier synchronization, (BARASGN, BARSYNC, and BARREL).
- debugging support (BUFTUNE, BUFPRINT, BUFDUMP, BUFUSER, and BUFENABL)

TSKSTART, which takes the name of a subroutine and a list of arguments to pass to that subroutine as its argument, is used to start execution of a task. TSKWAIT is used to wait for the end of execution of that subroutine. TSKTUNE can be used to change parameters related to the scheduling of tasks by the library level scheduler.

LOCKASGN is used to assign an integer variable the role of critical region lock. The critical region is entered through LOCKON. If the region is locked, the task is blocked, and the logical CPU may be rescheduled. Once the lock holder releases the lock with LOCKOFF, the task may continue. LOCKREL is used to indicate that the lock is no longer needed. Code within the library attempts deadlock detection and invalid lock use detection. Because locks are frequently used to implement critical regions which update a single value, a number of special purpose increment routines are provided.

EVASGN is used to assign an integer variable the role of event flag. When a task must wait for the event, it calls EVWAIT which blocks the task until the event is raised by EVPOST. After an event has been posted, any task which waits on it will return immediately until EVCLEAR is called. EVREL is used to indicate that the variable is no longer being used as an event flag.

BARASGN is used to assign an integer variable the role of barrier synchronizer. When the barrier is assigned, a countdown value is set. Each time a task calls BARSYNC, the task is blocked and the barrier counter is reduced. When the barrier counter reaches zero, all of the tasks which are blocked on the barrier are released, and the barrier counter is reset to the barrier value. This is used to implement rendezvous behavior.

3. UniCos Microtasking Background. Microtasking is currently available on the Cray 2 in a beta test version under UniCos 4.0. The version described here is preliminary, and probably will change in the future. Microtasking can be thought of as a form of multitasking which has been optimized for a particular kind of parallelism. If the work done on each trip through a do loop is independent of the work done on any other trip through that loop, it is possible to microtask that loop.

The microtasking implementation shares some features from the multitasking implementation. In particular, a group of shared address space processes are created by calling TSKSTART which invokes tfork, as described above. This invocation of TSKSTART is perfomed by the library, not explicitly by the Fortran program. Scheduling is handled in a different manner. Microtasking starts one process for each real processor in the system. When the code is not in a microtasked section, one process runs the program and the others are idle. When the code enters into a microtasked section, tasks obtain work to do under the direction of the master task. Each time a task completes, it requests more work, until all of the work is done.

There are also kernel extensions designed at reducing microtasking overhead. UniCos 4.0 has not been available long enough for a detailed analysis of these mechanisms, but microtasking overhead appears to be significantly less than multitasking overhead.

Microtasking is accomplished by inserting calls to subroutines into the Fortran code, as described above. In the current implementation, microtasking is accomplished by inserting compiler directives, in the form of Fortran comments. A preprocessor translates the directives into subroutine calls to the microtasking library.

Any subroutine which is to be microtasked must start with the `MICRO` directive, which simply tells the preprocessor that microtasking is occurring. Currently, microtasking provides three distinct ways to generate control structures for the program.

General parallelism can be accomplished by using `PROCESS`, `ALSO PROCESS` and `END PROCESS` directives. Several code sections which can be executed in parallel are started with a `PROCESS` directive, which indicates the start of a parallel section. Each new parallel section is started by the `ALSO PROCESS` directive. Finally, the parallel section is ended with the `END PROCESS` directive. It is possible to generate a parallel control structure with only one process, by using `PROCESS` and `END PROCESS` without intervening `ALSO PROCESS` directives.

Critical regions can be explicitly coded by the use of `GUARD` and `END GUARD` directives. `GUARD` n is used to indicate entry into a numbered critical region, and `END GUARD` n is used to indicate exit from that critical region. Different critical regions may be simultaneously active, but only one process may be in a region with a particular n value. These general mechanisms have not yet been evaluated.

The most common use of microtasking is in splitting do loops into parallel execution. This is done by placing the `DO GLOBAL` directive prior to the statement which starts the do loop. This causes loops to be parallelized as described above. Because do loops can be sped up by both parallelization and vectorization, the `DO GLOBAL` directive allows three modifiers which control the way in which work is split up by the processes doing the work.

The simplest of these is `DO GLOBAL LONG VECTOR` which divides the loop into processes of 64 iterations each and vectorizes the iterations. The number of iterations performed by a task is called the *chunking factor* and can be explicitly controlled using the `DO GLOBAL BY` n directive which causes n to be used as the iteration count, rather than 64; or the `DO GLOBAL FOR` n directive which causes n tasks of the appropriate number of iterations to be executed.

4. The program. A very simple 80 line program was obtained from [2], pages 334 through 335. This program numerically solves a two dimensional elliptic partial differential equation of the form:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + fu = g$$

where $f = f(x,y)$, and $g = g(x,y)$ are given continuous functions defined in some region R. $u = u(x,y)$ is the solution function whose values are only known at the boundary of the region. The program at hand solves the problem on the region $0 \leq x \leq 1$ and $0 \leq y \leq 1$, for $f = 2$, and $g = x^2 + y^2 + (xy+1)(xy - x - y)$, for which it is known analytically that $u = .5x(x-1)y(y-1)$.

After initializing the boundary conditions, the program calls the subroutine `SEIDEL` to perform iterations on the five point approximation of the system of equations arising

from the finite-difference approximation of the solution to the system:

$$u_{ij}^{(n+1)} = \frac{1}{4 - h^2 f_{ij}}(u_{i+1,j}^{(n)} + u_{i-1,j}^{(n+1)} + u_{i,j+1}^{(n)} + u_{i,j-1}^{(n+1)} - h^2 g_{ij})$$

It then computes the difference between the achieved solution and the correct numerical result and prints the difference.

This program was chosen because it was small enough for easy implementation, yet could be sized to consume enough CPU time to give interesting results, because it solves an easy to understand problem which can be numerically verified, and because there are multiple ways of transforming the code into a multitasked application. It was not chosen as an efficient method of numerically solving the Helmholtz equation.

FIG. 1. *SEIDEL*

```
      SUBROUTINE SEIDEL(AX,AY,NX,NY,HX,HY,ITMAX,U,X,Y)
C
C SEIDEL 5 POINT ITTERATION GAUSS-SEIDEL BLOCK DIAGONAL SOLVER
C FOR ELLIPTICAL PDE FROM CHENEY/KINCAID PAGE 334
C
      DIMENSION U(NX,NY), X(NX), Y(NY)
      F(A,B) = 2.0
      G(A,B) = A * A + B * B + (A * B + 1.0) * (A * B - A - B)
      START = CPTIME()
      H2 = HX * HY
      DO 3 K = 1, ITMAX
         DO 2 IY = 2, NY - 1
            DO 1 IX = 2, NX - 1
               V = U(IX+1,IY) + U(IX-1,IY) + U(IX,IY+1) + U(IX,IY-1)
               U(IX,IY) = (V - H2*G(X(IX),Y(IY)))
     +                    /(4.0 - H2*F(X(IX),Y(IY)))
 1          CONTINUE
 2       CONTINUE
 3    CONTINUE
      STOP = CPTIME()
      WRITE (6,100) STOP - START
 100  FORMAT(' TOTAL CPU TIME = ', F12.4)
      RETURN
      END
```

In its original form, (see figure 1) SEIDEL consists of a triple nested loop. The outermost loop is the pass counter, which controls the number of times the Seidel iteration is performed. The next level in controls the Y dimension of the subscript and the deepest level controls the X dimension. In the timing runs, the iteration is performed 25000 times on a 100 by 100 element matrix. The innermost loop is conditionally vectorized on the Cray 2.

Once the program had been entered and verified, it was modified to be vectorizable

on the Cray 2, most of the i/o statements were removed, and the final loop was changed to output only those points which were in error by a specified amount, which was set to 0.001 percent. By setting the number of grid points in the X and Y directions to 100, the program took about five minutes to run to completion on the Cray 2, and it was analyzed using flow trace. As to be expected, 99.998 percent of program time occurred in the inner loop of the `SEIDEL` subroutine.

The first attempt to multitask the subroutine consisted of unrolling the middle loop into four separate loops, one for each quarter strip along the Y axis. In this case, the four loops can execute separately of each other, and produce the same result as the initial application. This was done by writing a routine which replaced the second loop by four calls to `TSKSTART` and four calls to `TSKWAIT`. `TSKSTART` starts a subroutine which consists of the orignal inner two loops, with the original middle loop now an outer loop which performs one fourth of the work.

The second attempt to multitask the program was to only start four tasks, each of which invokes a complete copy of the Seidel routine on one quarter of the data. Rather than using task starts and waits to synchronize, this version uses barrier synchronization. As each routine reaches the bottom of its outermost do loop, it enters a barrier and waits for all other routines to reach the end of their loop. This *pseudomicrotasked* approach was an attempt to approximate the automatic approach which had been described for microtasking.

The third attempt to parallelize the program was to actually use microtasking to microtask the middle loop of the program. This was done by inserting a `DO GLOBAL` directive just before the beginning of this loop.

TABLE 1
Standalone

Program	elapsed time	CPU time	Speedup	MFLOPS
Single Tasked	229	229	1.00	18
Multitasked	171	550	1.34	23
Pseudomicrotasked	101	394	2.27	40
Microtasked	61	243	3.75	67

TABLE 2
Multiuser

Program	elapsed time	CPU time	Speedup	MFLOPS
Single Tasked	437	248	1.00	9
Multitasked	2085	2820	0.20	2
Pseudomicrotasked	953	1086.6	0.45	4
Microtasked	304	254	1.43	13

5. Findings. Table 1 contains the results from the standalone runs of the program. Table 2 contains the results from the multiuser runs. Standalone times were measured

running on the system console, with no other processes running on the system. They represent the best possible times. Multiuser times were measured running on a heavily loaded system during the middle of prime shift. An attempt was made to run the multiuser tests under the same load condition, so that they all represent the same overhead load for the system. Each code was run several times and the mean time is reported in Table 2.

Elapsed time is for only the SEIDEL routine, as measured by using calls to the system TIMEF routines from within the mainline program. CPU time is for the SEIDEL routine, as measured by the MTTIMES subroutine. Speedup is the ratio of the elapsed time for the single tasked time in a mode to the elapsed time for this version of the program in that mode.

MFLOPS is the measure of million floating point operations performed per second and was obtained by counting operations in the inner loop of SEIDEL and multiplying by the number of trips through the loop. For the test runs, NX and NY were set to 100, and ITMAX was set to 25000. The inner loop contains 10 add operations, 6 multiply operations and 1 divide operation which were counted as a total of 17 operations, so that the inner loop performs $25000 * 98 * 98 * 17 = 4081$ MFLOPS. This number was divided by the number of elapsed seconds to obtain the MFLOPS rate.

The poor standalone performance for the multitasked program is a result of choosing an inappropriate scheme for partitioning the problem. The program creates thousands of new tasks and the overhead of task creation dominates the amount of CPU time used to do the work of the program. This approach would only be useful if task creation overhead was much lower.

The poor standalone performance of the pseudomicrotasked version stems from the use of multitasking primitives to simulate the more efficient microtasking library primitives. A major portion of the difference comes from the way in which the CPU scheduler interacts with synchronization primitives. The key distinction between pseudomicrotasking and actual microtasking is the granularity of synchronization. The pseudomicrotasked version assigns each task a large fraction of the work, and synchronizes all tasks simultaneously at the end of a large fraction. The microtasked version assigns each task a smaller fraction of the work, and requires that tasks only synchronize with the master controller to obtain new work until all of the trips through the loop have been performed.

Because all tasks must synchronize in pseudomicrotasking, more time delay is encountered, and performance is lower. Much of this is due to the time the pseudomicrotasked code spends in the critical region updating the barrier information. The standalone pseudomicrotasked version used 165 seconds of CPU time for overhead, while the microtasked version used 14 seconds. This is a significant difference, since the original process only used 229 seconds of CPU time.

There are two apparent causes for relatively poor performance in multiuser mode. The first has to do with the way in which critical regions are used, and is quite apparent in the multitasked programs which perform frequent synchronization. The second, which is discussed later, has to do with CPU scheduling under load on a UniCos system.

There is a tacit assumption in the design of the task level library scheduler that "logical CPUs" actually equate to physical CPUs, and that when a multitasked job is scheduled, it is scheduled against the entire machine. This is not the way UniCos schedules processes. Once the processes have been created by `tfork`, they are scheduled in the same fashion as any other process on the system. The difference between the tacit assumption and the actual implementation is responsible for both of the problems.

The serious performance degradation occurs when tasks are required to synchronize through a critical region. In particular, it is possible for a task running on a CPU to enter a critical region and then become preempted. If one of the other tasks acquires a CPU and attempts to enter the critical region, it can become CPU bound while busy waiting on the semaphore, while the task in the critical region is not even executing. As the time spent in a critical region grows with respect to the time spent out of the region, the probability of this event grows, until in the worse case, a program can spend more time in the critical region spinlock than it spends doing work. This appears to be the cause of the poor multiuser performance of both of the multitasked applications. This problem is aggravated by the large number of synchronization events in each program.

There is a parameter to `TSKTUNE` which allows the user to control the amount of time that a process will spend executing the spinlock before it gives up control of the processor. This parameter is `HOLDTIME`, and CRI recommends that it be set to zero, to avoid this problem. However, it should be noted that the program times given here were obtained with `HOLDTIME` set to zero for both the standalone and multiuser runs, so it doesn't appear to solve the problem.

When a process runs in standalone mode, it is not competing for the CPU with other processes and it can utilize all of the CPU cycles. When it is running in multiuser mode, it is in competition, and will be preempted. In a compute bound system, the UniCos scheduler effectively divides the CPU cycles in a roundrobin fashion, so that each job will receive a fraction of the time equal to the number of CPUS divided by the number of processes in the system. Scheduling is done at periodic intervals, called clock ticks. The length of an interval is refered to as a scheduling quantum.

In Unix systems, it is common to call the effective number of compute bound processes the load average, and to note that the ratio in wall time between a multiuser run and a standalone run is approximately equal to the load average.

Because microtasking introduces more processes to do the same amount of work, microtasked wall time should be less than single tasked wall time by a factor of the number of tasks in both standalone and multiuser time. However, what was actually seen was that microtasked multiuser speedup was significantly less than expected. Detailed timing from multiuser runs shows that the microtasked job was rarely running on more than one CPU at a time, and effectively was obtaining no better performance than the single tasked job. For some runs, the job never obtained more than a single CPU, and so actually showed speedups of less than one, due do the overhead introduced by the microtasking primitives.

This is also due to the Unix scheduler. Under UniCos on the Cray 2, when a process from a job obtains a CPU no attempt is made to obtain CPUs for other processes in

that job. As time passes and the processes which make up the job are scheduled on and off of different processors, other processes come between them on the run queue. Eventually the job can reach a state where each process is preempted before the next process in the job can reach the head of the run queue. In this state, the job never actually runs on more than one CPU at a time.

However, the microtasked job should still show a throughput improvement because it is made up of more than one process, and so should still receive more CPU cycles as a result. Observation of a microtasked job running on the system showed that consistently one process received significantly more CPU cycles than the other processes. Typically, 3 of the 4 processes would be within one second of each other in CPU time consumed, and the fourth would have between two and three times as many CPU seconds as the others. Statistics gathered by the microtasking library show that this process is the microtasking master process, and that typically it also accepts twice as much of the work. It is not yet clear why the microtasking scheduler is not assigning the work evenly, but this is having an effect on throughput

6. Multiprocessor scheduling model. Much of the understanding of multitasking and microtasking behavior came from reading traces of program execution, sampling process status while jobs were running, and evaluating accounting and timing data made available from the system. This was sufficient to point out the problems, but did not provide a clear understanding of the causes. To better illuminate the causes, an animated model of multiprocessor scheduling in a compute bound environment was developed.

This is not a complete quantitative model of the UniCos scheduler, but rather a qualitative model of multiprocessor scheduling behavior under compute bound load. The model assumes that an ordinary compute bound process will continue to consume cycles as long as it has a CPU and will only give up the CPU when it is preempted. A multitasked job is assumed to contain work which can be evenly divided among the tasks which make up the job. It is assumed that work is divided into even length intervals and that the process must enter into a critical region after each work interval. Multitasked jobs are assumed compute bound, and only give up the CPU upon preemption. The remaining processes on the system are ignored, except for their effect on process preemption. When a process acquires a CPU it is assigned a quantum randomly chosen between 0 and the maximum scheduling quantum. This effect approximates preemption for I/O completion and other system activities.

It is possible to control the number of processors in the system, the number of processes in the run queue, the number of multitasked jobs in the system, the number of tasks in each multitasked job, and the maximum scheduling quantum. These parameters are used to approximate the machine being simulated. It is just as easy to model a standalone Vax as it is to model an eight processor Cray Y/MP under heavy load.

Makeup of the workload is controlled by varying the average length of a job, the work done in a single pass of a multitasked job, and the amount of time spent in the critical region. The first parameter controls the length of the jobs, the second parameter model the granularity of the tasks, and the third the overhead associated

with synchronization and communication.

Finally, it is possible to control two scheduling parameters. It is possible for a process which is waiting to enter a critical region to either hold onto the CPU forever, or to give it up after a specified number of ticks. It is also possible to introduce a scheduling modification called *Affinity*. Processes have an affinity for each other when they tend to be placed on the run queue near each other. In this implementation, affinity is accomplished by returning a process to the run queue in the position immediately following the last process in its job, rather than at the end of the run queue. If the job only has one process, it is returned to the end of the queue.

While the model is running, it provides two graphic displays, in two separate windows on a Silicon Graphics Iris. In the first window, the run queue and CPU states are represented graphically, with one box for each entry. Numeric values are displayed within the boxes, and color is used to indicate the ownership and state of the process. In the second window, three strip charts are drawn which show the degree to which multiple processors are used by a single process, the percent of time spent waiting to enter critical regions, and the overall throughput of the system.

The model keeps statistics on the number of jobs and tasks which have completed, the amount of time spent running work, in critical regions, and waiting to enter critical regions, the average throughput, and the degree to which one job uses multiple processors.

It is possible to reset the statistics and modify all of the parameters during the execution, except the number of CPUs. It is also possible to print a copy of the current summary statistics at any step in the model.

Several million seconds of CPU time have been modeled under various load conditions. Both the critical region problem and the difficulty in obtaining multiple processors can be clearly seen in the animated display, and in the resulting statistics. The model shows that not holding the CPU when waiting to enter a critical region greatly reduces the overhead from spinlocks. It also shows that the likelihood of having the problem is directly proportional to the relationship between task granularity and the length of time spent in the critical region. The likelihood of busy waiting goes up as the time spent in the critical region goes up and as the number of entries into the critical region goes up.

There is a tradeoff in using holdtime, since the released process goes to the end of the run queue, and if the time spent in the critical region is small, this can adversely effect the throughput of the multitasked jobs.

The model also shows that using affinity improves the likelihood of obtaining multiple processors at once, and also improves the throughput of the multitasked job, at the expense of the throughput of other jobs in the system.

The model does not explain the imbalance in load under microtasking.

7. Recommendations. Because it is difficult for the CPU scheduler to know when a process is in a critical region, little can be done about the problem of busy waiting when the process in the critical region is not in a CPU, other than using the `HOLDTIME` parameter to limit spinlocking, and using multitasking only when the amount

of work between synchronizations is very large. The amount of time actually spent in critical regions should be minimized. The overhead of starting tasks should be avoided by partitioning programs to generate a small number of tasks and then using synchronization primitives, rather than by synchronizing by creating new tasks.

The problem of obtaining multiple processors during multiuser time need only be solved if an installation wishes to benefit the users of microtasking by providing them with better throughput as a result of their effort to microtask their codes. If the run queue on a system becomes small enough to allow for idle processors during multiuser time, then microtasking will find the extra CPUs. Further, the overhead due to microtasking is small, so there is little cost of running with microtasking when multiple processors are not obtained.

To obtain the benefit of multiple processors during multiuser time, some modification can be made to the scheduler to increase the chance that the processes which make up a job are scheduled at the same time. A simple choice is to directly address the problem with multiprocessor scheduling. Since processes which are adjacent in the run queue are more likely to be scheduled so that they are on the processor simultaneously, a simple modification to the scheduler which recognizes tasks when they are preempted and places them on the run queue immediately behind the last runable task from their job will increase the throughput of multitasked jobs, at the expense of single tasked jobs.

More time is necessary to study the load balancing problem in microtasking, and its solution should also improve the throughput of microtasked jobs during multiuser time. As the implementers gain experience with microtasking, the overhead will be reduced, which will further enhance both standalone and multiuser speedup from microtasking.

REFERENCES

References.

[1] CHENEY, W., AND KINCAID, D.
Numerical Mathematics and Computing.
Brooks/Cole Publishing Company, Monterey, California, 1980.

[2] CRAY RESEARCH INC.
Cray-2 Multitasking Programmer's Manual SN-2026.
Cray Research Inc., Mendota Heights, Minnesota, 1988.

Unicos Fair Share Scheduler

Ralph Knag

AT&T Bell Laboratories
2F-217
600 Mountain Ave
Murray Hill, N.J. 07974
(201) 582-5291
rhknag@mhuxo.att.com

ABSTRACT

AT&T Bell Laboratories has modified Cray's Unicos operating system to incorporate a hierarchical Fair Share Scheduler that allows us to assign logical portions of the machine to different sets of users. This paper will briefly describe the Share software that served as the basis for this scheduler and then discuss the changes that we made to it for the Cray environment. We will also review our experience to date with this system. Our code served as a model for the Fair Share Scheduler that is to be included in Unicos Release 5.0.

UNICOS System Administration at the Ohio Supercomputer Center Tuning Considerations

By

Kevin Wohlever
Cray Research, Inc.

Abstract

General administration practices that work well with a small group of use rs, or machine, may no longer be valid on a supercomputer capable of supporting a much larger user database. As vendors add services to support more users in a common environment, ef forts must be made to introduce these services in the most efficient manner. Systems m ust be configured with the resource requirements of the services in mind, or overall system throughput and performance will suffer. The speed of peripherals (disk, tape), front- -end connections, applications, file-system, system configuration, and system design will all affect pe rformance.

This presentation / small paper will discuss the effects of a poor config uration. A brief discussion of other tuning considerations will be included.

The Ohio Supercomputer Center supports a large, diversified user communit y. With a Cray X-MP / 24 computer running UNICOS, the center supports user access throug h bitnet, DECnet, and TCP/IP. The Bitnet community can submit batch jobs to an IBM ma chine running MVS. The job is then sent to the Cray using the MVS station software fro m Cray Research. DECnet batch jobs are submitted to the Cray in roughly the sam e method, but in this case remote users run a version of the Cray VMS station on their loc al machines. A job is submitted on the local machine, routed to the attached VAX at the Ohio Supercomputer Center, and then submitted to the Cray. TCP/IP users can run interactively on the Cray.

Supporting the three different access methods requires additional softwar e to be run on the Cray. UNICOS is modeled after AT&T UNIX V.2, with extensions from V.3, B SD 4.2 and the vendor. These vendor extensions include UNICOS Station Call Protocol (USCP), Network Queuing System (NQS), Resource Management, On-line tapes, and Secur ity.

Because most of these extensions are implemented as daemons under UNICOS to keep kernel modifications to a minimum, a good deal of real memory can be used by these "system support" processes. The combination of these added processes and the div ersified user needs can cause a tuning nightmare.

Add to this a relatively small memory machine, and slow peripheral device s; the environment is ripe for severe problems. The Ohio Supercomputer Center suffered a nu mber of problems in its early stages because of these factors, problems that were hidden on larger memory machines, with faster peripherals. These problems were solved by a combi nation of software changes, and changes in administration techniques, most of which amount to what I like to call gross tuning methods. Some fine tuning methods implemented at the sa me time were cosmetic, and mostly done by accident.

Virtual Memory Extensions in TRACE/UNIX™

Patrick Clancy

Multiflow Computer
175 North Main Street
Branford, CT 06405

ABSTRACT

The TRACE/UNIX kernel is derived from Berkeley 4.3BSD. Because the TRACE must support a mix of both very large application programs and small UNIX utilities, it requires disk and memory management facilities which are more closely tied together than is possible with the 4.3 design. We have added file-system based backing store, and shared libraries based on global segments, as extensions to the 4.3 virtual memory system. This paper describes these extensions in the context of the particular requirements of the TRACE, and their impact on performance and system administration.

1. Introduction

The TRACE/UNIX kernel is derived from the Berkeley 4.3BSD release [7], and runs on the Multiflow TRACE™ series of Very Long Instruction Word (VLIW) machines. The VLIW architecture takes the RISC [12] approach to its logical conclusion by using many synchronized functional units to execute multiple operations during each instruction cycle. These operations are scheduled by the compiler from extended basic blocks called "traces" [8]. The machine is described in detail elsewhere [6]. The CPU architecture is mostly transparent at the operating system level, so that issues such as multiprocessor synchronization do not arise [5].

The TRACE architecture is distinguished by its simplicity and reliance on a "single" VLIW CPU. This CPU must handle both system and application programs. Since the TRACE is not a vector or attached processor, there is no front-end machine to perform operating system tasks or run utility programs. The advantage of this approach is that the performance benefits of the VLIW CPU are applied equally to system and application code. It also means, however, that the TRACE must support the very large programs intended for supercomputers while often contending with a multi-user workload having a high proportion of small, short-lived programs.

In this situation, management of disk space for both backing store and program files cannot be adequately supported using the facilities provided by 4.3. Since backing store in 4.3 is allocated from fixed-sized disk partitions, it is necessary to make worst-case assumptions about program size over the life of the system when these partitions are created. And since backing store is pre-allocated to processes at

UNIX is a trademark of AT&T Technologies. TRACE is a trademark of Multiflow Computer. DOMAIN is a trademark of Apollo Computer. VAX/VMS are trademarks of Digital Equipment Corporation.

run time whether or not it is actually used, maximally efficient utilization of this space is rarely achieved.

Since memory sharing between processes is not possible in 4.3, there is no way to eliminate the need to save redundant copies of library code and data. This problem is particularly applicable to the TRACE, because of the extreme degree to which it takes the RISC approach; best performance is frequently obtained through the use of aggressive compiler optimizations which result in code expansion.

To address these issues, we have added file-system-based backing store allocation and shared library support to the TRACE/UNIX kernel. Shared libraries have been part of all production TRACE kernels; file-system-based backing store will be available in a forthcoming release. This paper will discuss the design and operation of these facilities, and the particular aspects of our approach which are most closely tied to the requirements of a mini-supercomputer.

2. Background

Various operating systems have integrated disk and memory management to a high degree. The single-level store approach, first seen in Multics [3] and then many subsequent systems such as the Apollo/DOMAIN [11] and the 801/CPR [4], treats files as objects which are accessible through the virtual memory system. Single-level store has not necessarily meant, however, that ordinary files are used as backing store for program-created data. Our approach is much closer to that of VAX/VMS [10], which provides variable-sized paging/swapping files, but not as part of a single-level store design.

In the UNIX domain, both research and commercial kernels have provided significant new virtual memory capabilities relative to the original Berkeley design [2]. These often include shared memory as a basic feature, but the degree to which file-system and virtual memory resources are integrated varies widely. The Mach [13] and SunOS [9] kernels provide this sort of integration to some degree, including single-level store ideas. The Mach design provides for user-level paging data managers, which may presumably use any medium including normal files as backing store. However, the general practice seems to be that even UNIX variants with enhanced virtual memory systems still use fixed swap partitions for backing store.

With a general shared memory facility in the kernel, shared libraries can be implemented entirely outside of the kernel. This is the approach that has usually been taken with UNIX-based designs, such as System V [1].

2.1. Design Choices for a Mini-Supercomputer

Any virtual memory system improvement is potentially applicable to a broad spectrum of machines (as is UNIX itself). Our design decisions, however, have been guided by an overriding concern with performance and application program size.

It is also important that virtual memory facilities be easy to ignore, or easy to manage when necessary. The majority of TRACE users develop or run scientific/engineering applications, and the operation of the virtual memory system should usually be invisible to them. Where system administration is required (e.g., to allocate paging space on a disk), or the programming environment is affected (e.g., to set up a new shared library), the interface should be simple and

based on existing mechanisms. For example, standard utilities should yield useful information about new entities like paging files or shared libraries.

As noted, general inter-process shared memory is provided in some versions of UNIX. These usually allow sharing of pages, and allocation of mapped or shared areas within some larger region of the address space. In our case, a more limited form of shared memory was added to the kernel specifically for shared libraries. This is actually an advantage in some respects which were important to us: (1) General sharing involves more extensive kernel changes with a consequently less predictable performance impact. (2) More importantly, page-level sharing allows the possibility of aliasing of shared text addresses, which has an adverse impact on any use of virtual translation buffers or instruction caches; this is discussed later.

In certain cases we have traded off generality in the design of a feature for performance advantages or ease of use. For example, this was the case with our decision to statically link programs to shared library data addresses, and our choice of a global paging file strategy rather than mapped files to backup virtual segments. We have not restructured the kernel in any fundamental way, but rather have extended the existing base. This has allowed us to concentrate on dealing with the particular problems associated with running large programs in a multiuser UNIX environment.

3. Memory Management in TRACE/UNIX

The TRACE supports virtual memory via two translation look-aside buffers, for text (ITLB) and data (DTLB). Each TLB entry maps one 8K byte virtual page, so the 4K entries for each TLB map 32 megabytes of memory. The TLBs are accessed via a hashing scheme using high-order bits of the virtual address and a hardware process id (pid). Thus, use of a given virtual range by multiple processes and the kernel does not necessarily result in TLB collisions, and TLB entries are not flushed on context switch. A total of 255 hardware pids are available, with one reserved to the kernel. Shared process text segments have a single hardware pid. When a memory reference results in a TLB miss, "trap" code (which is normal TRACE code) handles loading of the TLB entry, or calls the kernel for a page fault. Since the trap code runs unmapped, a software-maintained table with 16 entries (each representing 256 megabytes) is used to keep physical page table base addresses and lengths for the currently running process.

Processes consist of multiple virtual segments, including at least the standard UNIX program segments (text, initialized data, uninitialized data, and stack), and possibly more for shared libraries. Text segments are generally shared among processes referencing them, and appear at the same virtual address in each (this is what allows the use of a shared hardware pid).

The page replacement and swapping policies are as provided by 4.3 [2]. Replacement is based on the global clock algorithm, with a few adjustments (discussed later) to support dynamic backing store management. The clock scans the *cmap* list of physical page frames for replacement candidates. A page clustering* mechanism is used to transfer multiple pages on a page-in or page-out operation; this identifies pages that are contiguous both within the virtual space and physically on the disk.

*Referred to as "klustering" in the sources.

3.1. Copy on Write

We have implemented copy-on-write mapping of data pages as a means of improving *fork* performance. A copy-on-write page is simply flagged as such (and made read-only) in the page table entry.

Page-outs of copy-on-write pages are not allowed, because (1) it is costly to maintain the information required to find all references to such pages; (2) a fork is often soon followed by exec'ing of a new image, eliminating the copy-on-write references just created; (3) the clock algorithm uses invalidation of page table entries, and hence TLB entries, to simulate LRU replacement in the absence of a reference bit, and this may lead to unnecessary TLB misses if there are many processes which are only reading these pages (the effect on the TLB of freeing shared text pages is not so bad, since the same hardware pid is used by all processes sharing them). We are saved from deadlock in extreme situations by the fact that swap-outs act to decrease the reference count on these pages, until they can be paged out.

4. File-System-Based Paging

To eliminate the constraints imposed by the statically sized and allocated swap space partitions of the 4.3 design, we have provided a mechanism by which backing storage for processes is allocated from one or more file-systems. The design goals behind this mechanism were: (1) to achieve performance comparable to the swap-partition method, (2) to support large processes with maximally efficient utilization of both memory and disk space, (3) to allow flexible partitioning of disk space between backing store usage and normal file-system usage, and (4) to do the above in a manner that is easy to administer and does not require interrupting normal operations to make major resource allocation changes.

It would be possible to make swap space allocation less static by retaining the idea of separate swap partitions, but adding some mechanism for dynamically varying their size. However, it would only be feasible to shrink them in general, since shrinking of a file-system partition to accommodate an expanding swap partition would usually require compaction of data blocks.

Our approach is to allow a group of paging files to be created with a system call; these files constitute a global backing storage pool which is shared by all processes (qualified by access rights, described below). Space is allocated on demand from this pool, in units of file-system blocks (16K bytes on current TRACE systems).

There are similarities between our approach and that of VAX/VMS [10]; it also uses "global" paging/swapping files which may each contain data from many processes, and new files may be created after the system is booted. In VMS a given process is allocated to a single paging file during the process' lifetime. Once a new paging file has been created, it cannot be removed until the system is rebooted.

4.1. Paging Files

A key aspect of space allocation is the fact that paging files are fully filled with disk blocks as far as the file-system is concerned. That is, these files have no "gaps" where physical disk blocks are not allocated. Furthermore, this allocation is fixed for the lifetime of the file, as truncation is not allowed. Special in-core maps of

these files are maintained by the kernel, to translate logical file blocks to physical disk block addresses.

This approach satisfies our design goals in two ways: (1) it makes allocation of a block for paging very fast, as no disk access is involved; and (2) it ensures that a known amount of disk space is available for paging, independent of any other activity that may be taking place on the same file-system (e.g., creation or expansion of normal data files), so that a system administrator may control space allocation on the disk. Note that if blocks were not pre-allocated in this way (by creation of fully-filled paging files), it would be very difficult to control disk space usage; e.g., a disk which appeared to have space for paging at one point could have been completely filled by the time that space was actually demanded. In exchange for this speed and ease of resource control, we do however sacrifice some amount of dynamic behavior which would result from using the file-system's pool of free blocks directly.

Paging file size is always an integral multiple of the file-system's block size (16K on current TRACE systems). There is therefore never any use of "fragments" which would require copying data on the disk.

4.2. Block Allocation

Paging file blocks are allocated from paging files only when needed to free up memory; i.e., when a process is swapped out, or due to activity of the page-out daemon. A single process can be paged or swapped onto many paging files (as discussed in the next section). However, it is desirable to limit the number of files used per-process, so that as many individual paging files as possible have a zero reference count at any given time, and hence can be deleted if necessary.* Therefore, a new paging file is chosen for a process only on allocation of that process' first paging block, or when the file from which its blocks are currently being allocated is full. The new file chosen is the one with the most free blocks, qualified by access rights (discussed later).

4.3. Block Deallocation

There is no pre-allocation of blocks when a process is created. This approach raises the spectre of deadlock, of course; but we avoid this possibility by means of a simple deallocation policy.

The problem can be solved by freeing allocated paging blocks (back to the paging file, not the file-system free pool) whenever the data they contain is brought back into memory consequent to a page fault; this is sufficient to avoid deadlock when the sum of the storage needed by all processes is not greater than the total amount of storage on the machine. This approach is not quite good enough, because (1) it deallocates blocks unnecessarily, and (2) demands may exceed the total storage on the machine. To handle the first problem, blocks are only deallocated on page-ins when available paging file space falls below a threshold value. To handle the second problem, processes are killed when demand exceeds total storage, until storage is sufficient.

*The possibility of performance gains from striping page transfers across multiple drives is handled transparently at a lower level, by using striped files.

The page-out daemon is largely where this deallocation policy has an effect. When it picks a candidate page to free, and the data must be written to disk, the page either will or will not have a block already allocated to it. If not, and there are no free paging blocks, then either (1) there are some other pages which already have blocks allocated (this is tracked by a counter), or (2) there are no such pages. In the first case, page-out simply gives up on the current page; eventually it will come around to one which already has a block allocated. In the second case, storage is exhausted, and a victim process is chosen to be killed. The victim is the process using the most total storage among a set of the most recently created processes.

The ability to dynamically free paging blocks on page-ins, along with the standard (page-out daemon) mechanism for freeing memory pages, helps to meet our design goal concerning memory plus disk utilization. A single process which uses N bytes of its virtual address space can run on a system with n bytes of physical (non-system) memory and m bytes of paging file space, if $(n + m) <= N$. In this way, we can attain maximum resource utilization, though obviously performance will degrade as this limit is approached due to increased paging activity.

4.4. Paging I/O

A single block backs up two pages on the TRACE. This, conveniently, was also the size of the page cluster we had been using for paging I/O under the 4.3 swap-partition kernel. So, page-ins bring in a block's worth of pages when possible, and page-outs may write out a block's worth of pages. Also, whenever a paging block is allocated this is noted in the page table entries or *cmap* entries for two adjacent virtual pages. All paging and swapping to disk blocks uses physical I/O, i.e. it does not use the kernel buffer cache which is accessed for read and write operations to ordinary data files.

4.5. Data Structures

Page-table entries for out-of-core pages have a 4-bit *pg_fileno* field, where 0 indicates a zero-fill page, 1 indicates a new text page (to come from the program binary file), and other values index a table in the process' u-area (per-process system data) if this is a data page, or a table in the shared text structure if this is a text page. The local table entry references a global paging file table (*pfiletab*), as well as a list of blocks the process/text currently holds from the specified paging file. The block lists reside in paging file maps (one per file), which map logical file blocks to physical disk addresses; blocks not in use are on a free list. The in-core process table entry also references a paging file (usually the same as one of the other paging files used by the process), which stores page table and u-area data blocks when the process is swapped out. Thus, a single process may have allocated blocks from up to 15 paging files, and the maximum number of paging files on the system is limited only by a kernel parameter. Since paging files are referenced from page-table entries, sequential virtual pages may go to different files (qualified by the fact that a single paging block backs up multiple pages).

With each process/text referencing lists of blocks it uses, freeing blocks on process exit is easy, and is done in constant time regardless of the number of blocks allocated. Freeing blocks on page-in is not as easy, but is only done when available paging space is low.

The *pfiletab* entries contain information about paging files, including a *vnode* pointer (file handle used by kernel file-system routines), a pointer to the in-core paging file map and list of free blocks from this map, a count of free blocks, and a reference count (number of processes using the file). When a new paging file is created, a new table entry and file map are allocated; when the file is unlinked, these structures are deallocated when the reference count goes to zero.

When a virtual page resides in a paging block on disk, the corresponding page table entry contains the physical disk block address. When a virtual page resides in memory but there is also a paging block allocated (category (3), above), the *cmap* (map of physical pages) entry saves the page table entry containing the disk block address of the paging block; this saved entry is simply re-instantiated when the page goes back out to disk. Thus, the physical disk block address of an out-of-core page is always directly available in the page table entry, and the per-process swap maps used in 4.3 are not needed.

4.6. Shared Text

A further saving is realized for text pages; they normally do not have paging space allocated at all. When one of these is paged in, the fill-on-demand page table entry (which has *pg_fileno* = *1* and a physical disk block address from the program file) is saved in the *cmap*, and may be re-instantiated if the page is freed.

This differs from the 4.3 practice of always allocating swap space for text, which allows pages for cached texts to often be found in a swap partition rather than the program file. With our ability to save the fill-on-demand page table entries, it is just as fast to bring in these pages from the file-system (see performance data, below). Paging block allocation for text starts when the text is written via *ptrace* (e.g., running under a debugger).

4.7. Interface and Administration

Paging files are created by a utility at boot time; this invokes the *fswapon* system call, which opens or creates the file, allocates all its blocks, and sets up the kernel's map of these blocks. The file's *inode* is flagged to indicate that it is special, and cannot be removed or changed in size by any means other than further *fswapon* calls.

Initial paging files are set up during initialization for multi-user mode (via the */etc/rc* file); before this, the system is in old-style swap-partition mode. This is necessary, for example, so that file-system consistency can be checked before paging files can be created or used. Processes created before paging file mode is enabled will continue to use swap partitions during their lifetimes; normally these processes will require very little swap space, so we keep only a small swap partition by default. (We also use this partition as a "miniroot" file-system area when booting from tape.)

New paging files may be created at any time (by the super-user). A paging file may also be removed using *fswapon*, which causes it to be immediately unlinked, but it will not actually be freed until all its blocks are no longer in use for paging. The point at which this will happen might in general be hard to predict, since any processes might be using these blocks. However, a list of processes using a given paging file can be obtained from the kernel; unlinking the paging file and then awaiting termination of these processes will free the space. This can be done even

if there is currently paging activity on the system, since the file is no longer accessible for further paging use.

This method can't guarantee that a particular paging file can be freed, e.g. if an idle shell process is swapped out to it. There are reasonably simple approaches which could force such processes to give up all their paging blocks without being killed; this is left as a future enhancement.

We believe that the use of global, visible, and fully-allocated paging files provides the necessary basis for easy administration and resource control. Since paging files are just like other files (given the restrictions on deletion and truncation), standard utilities like "ls" and "df" can give information about paging space. We have also enhanced "pstat" to provide a list of paging files and usage information. There are many other user-level facilities which could be provided; e.g., to automatically create new paging files when the space was needed, or delete them when not needed.

4.7.1. Private Paging Files

It is occasionally desirable to provide private paging space to a user or group. For example, it may be necessary to prevent arbitrary processes from using paging space which has been set aside specifically to run a particular program. A typical situation might involve the creation of a very large paging file which will only exist for the duration of testing or debugging work on this program, and which needs to be deleted shortly thereafter.

Rather than enforce some sort of quota or limits system, which would be difficult to track and would not solve this problem particularly well, we have provided simple rules which are followed when the kernel allocates paging blocks: (1) a process can only receive paging blocks from paging files for which it has read permission; (2) among this set of accessible files, the one with the minimum permissions relative to the process is picked. (Blocks are always allocated on behalf of some process or text, even though the page-out daemon is examining the global set of physical page frames.) For example, if there are two paging files with free blocks, and one has world read access while the other has only owner read access, and the process user id matches the file owner id, the latter file will be used even if the other file has more free blocks. Thus, a private paging file can be set up by simply giving it the appropriate owner id and access rights.

One exception involves texts, which may be shared by processes with different user and group affiliations. In this case, world read access is required. However, even with large programs text size is typically not a concern, and we only allocate paging space for *ptrace*'d texts anyway.

4.8. Performance

It might reasonably be expected that use of file-system blocks for paging would have an adverse performance impact, since these blocks will not be physically contiguous as often as they would be if allocated from within swap partitions, using the 4.3 algorithm which increases allocation size by powers of two as segment size increases. Also there should be a cost associated with the on-demand allocation of blocks. However, we have found that paging/swapping performance is about the same as before, although overhead is distributed differently.

The results of a paging test are shown in Table 1. This test runs many large and small programs concurrently, and is both CPU and I/O intensive (it is normally used as part of our system validation test suite). The first two columns show data obtained from running with three paging files on three different disks, vs. three swap partitions on the same three disks. All other conditions are precisely the same across runs.

measure	swap partitions	paging files	large clusters
pages in	12769	12262	14130
pages out	11179	11224	11732
pages freed	14878	15963	15674
swap outs	36	32	40
drive 1 seeks	6524	7021	6185
drive 1 cyl./seek	144	138	165
drive 2 seeks	12330	11186	12124
drive 2 cyl./seek	234	244	238
drive 3 seeks	4052	6268	4928
drive 3 cyl./seek	113	113	113
interrupts	32761	33734	33049
elapsed time (sec)	1878	1864	1883
user time (sec)	1561	1562	1567
system time (sec)	161	160	159

Table 1.

The measurements include page-out daemon activity (pages paged in or out, pages freed), swap-out activity, disk activity (seeks and average travel, shown as cylinders traversed per seek), and times. Both elapsed and CPU (user + system) times are about the same in these two runs. Drive 2, where most of the user file I/O was being done, shows fewer seeks with paging files and slightly longer average travel, since paging blocks are mixed with ordinary data blocks throughout the partition. The other drives were used mostly for paging, and show more seeks with paging files since the blocks are more widely scattered, but average travel is about the same. Note also that the page-out daemon was able to run somewhat faster with paging files, resulting in fewer swap-outs.

It was mentioned earlier that the pre-paging cluster size we had been using was fortuitously the same as our file-system block size. What if this were not the case? With contiguous blocks in swap partitions there might be some gain from using a larger cluster size. However, with relatively large (8K) pages, the point of diminishing returns is rapidly reached with increasing cluster size. The third column of Table 1 shows the same test run with swap partitions and the page cluster size doubled to 32K. The overall times are slightly worse, while the disk measures are mixed.

5. Shared Libraries

In contrast to the backing store allocation problems which are most apparent when running single very large programs, the need for shared library support results from the large collection of small utility programs which are available under UNIX. On the TRACE, the effect of duplicating standard library code for every program

that uses it, on disk (as part of the saved program file) and in memory (as part of the running program image), is particularly noticeable. This is because of code size expansion as a side-effect of the large number of optimizations performed by the compiler to keep the VLIW functional units busy. The primary causes of this expansion are generation of compensation code associated with traces, loop unrolling, and procedure in-lining; altogether an expansion factor of about 3 relative to VAX code size is the usual result [6].

There is also a performance issue, in that there is a cost associated with having to page-in code and data which may already be in-core for another process.

Our goals for a shared library mechanism were: (1) negligible run-time overhead, e.g. no run-time linking; (2) source level transparency, such that source code changes not ever be required in programs using libraries or the libraries themselves, and programs never have to be re-compiled (only re-linked) to use shared libraries; (3) the ability to update libraries with new versions, without recompilation or even relinking of existing user programs; (4) the ability to support multiple libraries (including use of multiple libraries by a single program); and (5) support for arbitrary user-created libraries.

The basic approach is to set aside a portion of the address space to be globally shared by all processes using shared libraries, and to reference shared library text via jump tables which are mapped into a non-shared part of the address space. All address resolution, to jump table addresses for text and fixed addresses for data, takes place at link (*ld*) time; when a program is executed it is attached by the kernel to whatever shared libraries were specified at link time (which are listed in the executable file header), these libraries already having been "installed" on the system in the manner described below.

5.1. Process Types

We have three categories of processes, distinguished by the magic number at the start of the executable file: (1) processes which do not use shared libraries; (2) processes which use shared libraries; and (3) shared libraries. (Within the first category, we of course support the standard types of demand-paged, non-demand-paged, and old format single segment. Processes in the other categories are always demand-paged). Processes in the first category do not have access to the global shared library address space segment; their text is mapped starting at zero. Processes in the second category access this global segment, via a single shared page table. All process types (including shared libraries) operate within a full 32-bit address space.

Shared library processes (category (3)) are real processes which execute an initialization sequence before blocking with most signals masked. This has several advantages: (1) A library is a type of object (process) that the kernel already knows about. Although of course there are many activities, such as paging and exec'ing, which require that the kernel do things somewhat differently for a shared library, in general it was not necessary to add an entirely new mechanism for "installing" and "removing" a library since these operations correspond to exec'ing and killing a process (and this is exactly what is done in our implementation). For example, a library can be started at any time by someone with superuser privileges, by executing it as a command. (2) Standard utilities like "ps" show shared libraries,

which also contributes to the perception on the part of users that no entirely new mechanism has been added. (3) Library processes can perform any sort of initialization activity, such as announcing that they are running.

It has been essential to have some programs which do not use shared libraries. For example, it is impractical to allow breakpoints to be set in shared library text while debugging. Also, the "init" program (process 1), which starts the default system libraries, must itself be built without shared libraries. Our C compiler currently links without shared libraries by default, while the Fortran compiler does the opposite. However, all the utility programs that are part of our standard operating system distribution are built with shared libraries.

5.2. Shared Segment

The global shared segment is mapped at zero, with private space starting at 256 megabytes. Shared library processes exist within the shared segment, primarily to have their text and data mapped by other processes. This segment is implemented by loading the physical address of the shared page table into the first entry of the segment table used by the trap code (see above) for page table access. Since this table is software-maintained, we can easily change the size of the shared area (in multiples of 256 megabytes) should it become necessary.

A given shared library therefore appears at the same virtual address to all processes using it. This eliminates any possibility of aliasing of shared text addresses, which allows the potential sharing of entries in the virtually-addressed processor instruction cache, and the ITLB.

It should be noted that if a general page-granularity sharing facility were used, there would be no straightforward way to take advantage of hardware translation or instruction caches to support shared libraries. This is because there would be no easy way to switch to a shared hardware process id on access to an arbitrary virtual page address range (this statement is made in the context of the TRACE hardware-pid mapping mechanism, but is really more general; changing the hardware translation context on access to a new virtual page would be expensive no matter how it was done). On the other hand, segment-based sharing where aliasing is not allowed does allow this hardware support.

Shared libraries constitute a resource which is distinct from but not orthogonal to the existing shared text facility. All processes using a given text also see the same libraries, at the same virtual address; but all users of the same library do not have the same text. So, with only one instruction cache and one ITLB, each entry is tagged with a hardware pid representing the text (not the library). It is a relatively simple hardware extension to provide a shared pid to be used by the translation hardware when the shared segment is referenced. (However, this is not done by current TRACE hardware. The text pid is used to access library code, but this only causes a number of duplicate ITLB/cache library entries on the order of the number of texts using the library, rather than the number of processes using it.)

One peculiar effect of shared libraries is important to consider when libraries are large. While standard archive-type libraries contribute only code which is needed by the program at link time, a shared library has the effect of separating functions that would otherwise be loaded contiguously into fewer pages from an archive-

type library, in effect bringing unneeded code and data into a process' address space. This means that the TLB miss rate should increase when shared libraries are used. (The effect on the page fault rate is negligible. For text, pages for popular functions will be referenced often, and so will tend to always be found in-core. For data, library pages are mapped copy-on-write, and the physical data pages are all read in when the library is started.)

This effect is reduced by the large page size and number of TLB entries, Also, when a shared library is built its constituent functions are loaded in an ordering based on caller-callee relationships, to yield the maximum locality of reference at run time. The result is that in practice increased TLB miss due to the use of shared libraries is not significant.

5.3. Jump Tables

In order to satisfy our requirement that new library versions can be created which do not invalidate existing executable files, a jump table is associated with each library. Each jump table entry is a branch instruction to a library function entry point. User programs are linked to branch to jump table addresses rather than directly to library code (and library function symbols, e.g. "printf", are set to refer to these table addresses, with new symbols created for the actual library code). A new library is said to be compatible with an older version if its jump table entries are a superset of the older version, and the entries which they have in common remain at the same offsets in the table, and the absolute addresses of global data symbols remain the same. Programs linked with the older library will still run using the new library. We have added support in the linker (*ld*) for creation of compatible libraries when possible, given an old library as a template.

Jump tables are mapped copy-on-write in the address space of the user program. This allows private versions of library functions to be substituted for the standard versions. Any function represented in the jump table is referenced *only* through the jump table, even from within the library. So, the address of the private version of a function can simply be written by the program into the appropriate jump table slot, which will cause a private copy of that jump table page to be created, and all further references to access the private version. This obviates the need to make source code changes in the library to handle this situation. This facility is not often needed, however, so most programs will share the jump table pages of the library itself, and overhead to set up this mapping at exec time is negligible. The ability to use shared physical memory for jump tables saves approximately 32K bytes for each process, with our standard C + Fortran library.

5.4. Address Space Allocation

Because shared library processes reside in a global shared segment, they may not overlap in this space. When a new compatible version of a shared library is created, a new text start address is chosen so that this library will not overlap with any old versions. This way, the new version may be installed and tested on a system which is still running an older released library (see the discussion of name binding, below). With a 256 megabyte shared segment, there is plenty of virtual space in which to run multiple versions of multiple libraries.

Both jump tables and library data are mapped as copy-on-write into reserved areas in the private process address space, below program text and above the stack respectively. The size of these areas is easily changed by rebuilding the kernel.

Shared libraries use up a small portion of the 4 gigabyte virtual address space available to a process. However, they only use areas at the extreme ends of the virtual space, to allow user program data to grow contiguously until the upper bound (stack area) is reached. It was specifically our concern with the support of very large programs that led to this design, which is in contrast to other UNIX shared library implementations. A shared library using process on the TRACE may consume approximately 3.8 gigabytes of non-library text plus data (including stack).

5.5. Name Binding

Libraries have symbolic names, which are the same as the last component of their pathname; and version numbers, which are split into two parts (major/minor). Programs which use shared libraries contain all or part of the library pathname (at least the last component) in their executable file header, along with a version number. When such a program is exec'd, the kernel extracts the library symbolic name from the header and looks it up in a table of running libraries, requiring a match on this name and major version number. Different major numbers for libraries with the same name indicate incompatible versions of the same library. If the kernel finds multiple matches on the name and major number, it looks for an exact match on the minor number; if no match is found, it uses the library with the lowest minor number greater than the one requested, indicating the oldest compatible version which is newer than the one requested. This means that when a new library version is started on a system, it is effectively masked by any older compatible libraries until these are killed. However, a program which has been linked specifically with the new library will use it.

This scheme allows us to deal easily with the issue of library testing and release which was mentioned earlier. A new untested, but compatible, library version may be started on a machine without affecting released utilities or other programs linked for an older version. New test versions of utilities may be linked with the new library, and will automatically be bound to this library when they are run (since they have the same old major version number and new minor version number). Once testing is finished, any older libraries with the same major number may be killed, and all programs which need this library will now start using the newer version.

5.6. Linking

Shared libraries are created by giving appropriate switches to the linker (*ld*), which creates an *a.out* format file, with the jump table occupying the start of the text area. The library file may then be specified on the command line to another link operation, which causes information about the library (name and version number) to be placed in the output file header, and references to library symbols to be resolved; library text and data are (of course) not included in the output. Note that the same library executable file, with text (including jump table), data, and symbol table, is used both for linking of other programs and as the executable library program. This is the simplest approach administratively (since there is just

one file, the *a.out*, representing the library), and is congruent with our notion that a shared library is just a type of program object rather than a new type of object.

6. Conclusion

The TRACE/UNIX virtual memory system has not been overhauled as severely as that of some other BSD-based kernels. However, the enhancements made to date have yielded perhaps the best value relative to implementation effort of any of the commonly considered changes (shared memory, single-level store, etc.), given the sort of job mix commonly encountered on the TRACE. At the same time, these enhancements are by default completely transparent to applications programs, and do not introduce any incompatibilities with the 4.3BSD interface.

7. References

[1] Arnold, J., "Shared Libraries on UNIX System V," *USENIX Conference Proc.*, Summer 1986.

[2] Babaoglu, O., W. Joy, "Converting a Swap-Based System to Do Paging in an Architecture Lacking Page-Referenced Bits," *Proc. Eighth Symp. on Operating Systems Principles*, Dec. 1981.

[3] Bensoussan, A., C. Clingen, R. Daley, "The Multics Virtual Memory: Concepts and Design," *Commun. ACM*, 15:5 (May 1972).

[4] Chang, A., and M. Mergen, "801 Storage: Architecture and Programming," *ACM Trans. on Computer Systems*, 6:1 (Feb. 1988).

[5] Clancy, P., B. Cutler, J. C. Dodd, D. Gilmore, R. Nix, J. O'Donnell, C. Ryland, "UNIX on a VLIW," *USENIX Conference Proc.*, Summer 1987.

[6] Colwell, R., R. Nix, J. O'Donnell, D. Papworth, P. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *Second Int. Conf. Arch. Support for Prog. Languages and Op. Systems*, Oct. 1987.

[7] Computer Systems Research Group, "4.3 Berkeley Software Distribution, Virtual VAX-11 Version," *Dept. of Elect. Eng. and Comp. Science*, University of California, Berkeley (April 1986).

[8] Fisher, J., "Trace Scheduling: A Technique for Global Microcode Compaction,", *Trans. on Computers*, v. C-30 (July, 1981).

[9] Gingell, R., J. Moran, W. Shannon, "Virtual Memory Architecture in SunOS," *USENIX Conference Proc.*, Summer 1987.

[10] Kenah, L., and S. Bate, *VAX/VMS Internals and Data Structures*, Digital Press, 1984.

[11] Leach, P., P. Levine, J. Hamilton, B. Stumpf, "The File System of an Integrated Local Network," *Proc. ACM Computer Science Conf.*, New Orleans (March 1985).

[12] Patterson, D. and C. Sequin, "A VLSI RISC," *Computer*, 15:9 (Sept. 1982).

[13] Young, M., A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proc. Eleventh ACM Symp. on Operating Systems Principles*, Nov. 1987.

Memory Management in Symunix II: A Design for Large-Scale Shared Memory Multiprocessors

Jan Edler, Jim Lipkis, and Edith Schonberg[1]

Ultracomputer Research Laboratory
Courant Institute of Mathematical Sciences
New York University
251 Mercer St.
New York, NY 10012

ABSTRACT

While various vendors and independent research groups have adapted UNIX and other operating systems for multiprocessor architectures, relatively little work has been done in anticipation of the software requirements of very large-scale shared memory machines containing thousands of processors. Programming environments for these machines must exploit multi-level memory and cache hierarchies so as to obtain the maximum performance available from the hardware architecture. Several years of experience at New York University with parallel systems and programming environments have led to a memory management design emphasizing flexible control over arbitrarily complex patterns of memory sharing and cacheing. This design integrates user memory management, virtual memory management (swapping/paging), program debugging and monitoring, the file system and buffer cache, and checkpoint/restart in a synergistic manner that extends the existing strengths of UNIX. It is targeted for two research multiprocessor prototypes, the NYU Ultracomputer and IBM RP3.

1. Introduction

Symunix II is a 4.3 BSD UNIX[2] compatible operating system being designed for highly parallel MIMD shared memory architectures, specifically the NYU Ultracomputer [Got87] and the IBM RP3 [PBG85]. Like Symunix I [EGL86], Symunix II is fully symmetric; i.e., no processor plays a distinguished role. Since scaling to a large number of processors (up to several thousands) is a central objective, serial bottlenecks are avoided wherever possible by using non-blocking parallel algorithms and data structures both of which are based on fetch-and-add [GLR83]. A key difference between Symunix I and II is in the area of memory management. Symunix I remains close to the V7 model; the Symunix II memory management design, which is the topic of this paper, incorporates a more flexible virtual

[1] This work was supported in part by I.B.M. under joint study agreement N00039-84-R-0605(Q) and in part by the Applied Mathematical Sciences Program of the U.S. Department of Energy under contract DE-FG02-88ER25052.

[2] UNIX is a registered trademark of AT&T.

memory model and an integrated set of novel features to support large scale parallel computing.

By now, many UNIX-based systems have been enhanced or redesigned for multiprocessors, and support a variety of parallel processing models [BO87, BW88, HK86, ME87, TG88, TR87]. Popular extensions include lightweight processes in the kernel as the unit of scheduling, and shared memory capabilities. Lightweight processes present a parallel programming model in which all resources are shared, including the entire address space [HK86, TG88, TR87].

Within the framework of a more standard UNIX process model, in which each process has private resources and its own address space, other explicit mechanisms for sharing have been designed. These include sharing designated sections of memory via inheritance (over **fork** system calls or the equivalent) as in SunOS [GMS87], Xylem [ME87], and Mach [TR87]; specification of virtual address ranges to be shared across an entire job, as in Xylem; sharing global memory regions as in System V (where the name of the region is a user-chosen number); sharing memory by using the file system name space to name shared regions, and file mapping to define the regions, as in SunOS and Dynix [BO87]; and the use of an interface for shared memory object management outside the kernel, as provided by the external pager interface of Mach. The latter three mechanisms facilitate sharing among processes not related by **fork** alone.

In Symunix II, the unit of scheduling is a full-featured UNIX process. This model is preferred over the lightweight process model both because of the expense of scheduling lightweight processes in the kernel and because of the desirability of per-process private resources (elaborated in Section 2.1). Parallel applications often require application-specific or language-specific task scheduling, and general-purpose, low-overhead tasking packages can be provided in user mode.[3] Within this process model, parallelism is created with **spawn**, a parallel generalization of **fork**. (Parallel programming may be characterized as the one area where **fork** is most often *not* soon followed by **exec**!) We refer to the set of processes related by **spawn** but not **exec** as a *family*; thus members of a family are those that execute the same object file, and a family intuitively is a parallel program. Memory sharing may be provided by inheritance over **spawn**. But sharing via inheritance alone is not sufficient for parallel programming. Therefore, shared memory is also provided through file mapping and the file system name space.

While these shared memory mechanisms are similar to those used in other systems [TR87, BO87, GMS87], the Symunix II memory management design is novel in a number of significant respects, in particular, in the integration of features. The file system serves as backing store for the entire process image (as in Sprite [OCD88]); file mapping is the *only* means by which a process may obtain memory. The apparent performance penalties are mitigated through use of several devices and optimizations, while library packages provide a compatible and easy to use interface.

Like other recent redesigners of the UNIX memory management subsystem, we are striving for enhanced performance through modern memory management

[3] See [ELS88] for further discussion of these process model issues.

techniques; clean interfaces between the memory management component and the other levels of the kernel; and, particularly, portability across architectures. Our initial implementation is targeted to three machines with highly dissimilar memory mapping designs: the Ultracomputer prototype known as Ultra II, based on MC68010/MC68451 processors (variable-length segments); the RP3 (fixed-size pages, mapping using segment and page tables); and the IBM RT PC, our token uniprocessor (fixed-size pages, mapping using an inverted page table).

The remainder of the paper is organized as follows. Section 2 outlines the parallel processing requirements that most influenced the design. Section 3 describes the user's memory model and the system interface. Some aspects of the internal design, including performance issues, are presented in Section 4. The concluding Section 5 gives a status report.

2. Memory Management for Parallel Processing

While many UNIX environments are dominated by highly interactive, short running, non-numeric types of programs, the parallel processing world is dominated by large, long-running, scientific or other compute-intensive applications. Often a single job, consisting of many subtasks or processes, will want to take over an entire computer system for a large chunk of time, with little interference from the operating system. This perspective has sometimes resulted in the viewpoint that there should be minimal operating system services provided on parallel machines — perhaps little more than a batch scheduler.

Our viewpoint is that shared memory parallel machines should have general purpose multi-programming operating systems, and *general* facilities that take into account the special needs of parallel computing. In the area of memory management, there are a number of issues beyond the mere presence of shared memory that must be addressed. We detail them in the following.

2.1. Shared and private memory

Our parallel program model does not preclude hardware-isolated (i.e., separately mapped) private memory. Besides the fact that it is necessary for UNIX compatibility, there are several reasons why private memory may be useful:

- Private memory provides protection from wild stores in which non-shared variables might be clobbered asynchronously by other processes, causing obscure bugs in supposedly safe sections of a parallel program.
- For certain processor architectures, absolute addressing often results in more efficient code. In programs with a single level of parallelism, private variables may be assigned identical virtual addresses that are individually mapped to private memory regions, thus allowing absolute addressing modes.
- Absolute addressing lessens the need for reserved base registers, thus again improving the efficiency of generated code for user programs. This has arisen in the development of microtasking environments.
- Standard UNIX compilers and libraries assume absolute addressing for statically allocated variables. Although a secondary issue, such software is less portable when private memory is unavailable.

The advantages of absolute addressing stem from the fact that the "registers" of the memory management hardware can be used in effect as extra pointer registers by the executing program.

Sharing via selective inheritance over **fork** and **spawn** is a particularly useful mechanism for parallel processing, as it is a simple and clean way for parallel processes executing the same program text to have both shared and private data. Why are other sharing mechanisms needed?

- The inheritance mechanism does not allow shared memory to be dynamically allocated by a process of an existing family, after a **spawn** has already been executed.
- More sophisticated parallel applications can be larger than a single family, and memory sharing may be desirable between more remotely related processes. For example, a facility for inter-family shared memory allows for IPC implementations entirely in user mode.

Dynamic sharing, and a scheme for naming memory regions, are necessary for addressing these requirements. We have chosen to use the file system for this purpose because it is a rich, structured naming scheme that allows nested scoping of names, provides protection, *and* because it builds on the existing naming conventions for objects in UNIX.

2.2. Flexible low-level interface

Large-scale shared memory MIMD machines will have large virtual and physical address spaces and a hierarchy of memory, possibly including one or more levels of cacheing.[4] On such machines, each region of a process's virtual memory has several attributes, including read/write/execute protection, shared/private scoping, cacheable/non-cacheable access, and local/global physical memory. For dealing effectively with different kinds of memory, the standard UNIX model of fixed text, data, and stack segments is entirely inadequate, being both too high level and too rigid. Because of the wide number of varying requirements of parallel applications, and the many ways a large virtual address space may be split into different flavors of memory, it is too restrictive for the kernel to assign fixed virtual addresses. It is also undesirable for the kernel to attribute meanings to various memory regions. For example, a parallel programming environment may not have use for the standard type of stack that is provided for sequential C, but rather a cactus stack, or multiple stacks allocated within a general purpose heap.

For these reasons, the Symunix II kernel memory management interface is both low level and flexible, and allows for user specification of virtual and physical attributes. The attributes are independent and orthogonal: it should not be assumed, for example, that shared memory is always global (a process mailbox

[4]The RP3 has shared local memory, as well as shared global memory, accessed through an interconnection network. The local memory is accessed sequentially, while the global memory is interleaved. Local memory on the Ultracomputer is not sharable. Both the RP3 and Ultracomputer have private processor caches which may cache either local or global memory. Another approach to the memory hierarchy is used in Cedar [KDL86], which has shared global memory and private cluster-level memory for clusters of eight processors.

may be shared and local), nor that private memory is always local (in the presence of processor context switching, this may be undesirable). Virtual address space management is the responsibility of applications, libraries, and language environments. This does not necessarily burden the programmer — standard libraries are provided — but rather allows for a diversity of environments.

2.3. User control over cacheability

Large-scale shared memory MIMD computers with processor caches cannot provide automatic cache coherence in hardware without serious performance penalties. Currently available cache coherence technology, based on inter-cache communication over a bus, does not scale up to highly parallel configurations which are interconnected via multi-stage networks. Proposed directory schemes [CF78, AB85] may provide a partial solution for large systems, but the expense is sufficiently large that automatic coherence (without software control) appears undesirable. Currently, both the Ultracomputer and the RP3 allow the operating system to suppress cacheing for certain areas of memory; the cacheability option is specified as a component of the virtual-to-physical memory mapping.

In theory, cache coherence for user programs might be obtained in several ways:

a) Enforce a policy in the operating system whereby all shared memory that is not read-only is non-cacheable.

b) Implement software cache coherence in the operating system. The kernel would trap stores to shared cacheable memory, and immediately notify other processors to invalidate their caches.

c) Do not enforce cache consistency in the kernel, but provide for user mode control over cacheability.

Option a) is not adequate, because there are a significant number of parallel applications for which the ability to cache read-write shared memory is important for performance. A typical example is a parallel program in which each process computes values within a section of a grid represented as a multi-dimensional array. Such applications exhibit great locality of reference, with occasional non-local references to neighboring grid sections. Both for ease of programming and compilation, and to allow for dynamic adjustment of the grid and temporary access of neighboring grid sections, the grid array should be allocated in shared memory. Moreover, because of the local nature of the computation, efficiency suffers greatly if shared writable memory is not cacheable. Chaotic relaxation [Ba78], in which consistency can be eased over several iterations, allows for even more cacheing.

Option b) is very inefficient, because the operating system must intervene on each store to a shared variable and because some stores to shared memory may not require invalidation of other processors' caches anyway, as the grid example illustrates. Furthermore, this is a serial solution which does not scale well. Ultimately, only the application programmer, or optimizing compiler, knows when cacheing is safe and when the caches must be invalidated or flushed. There would be little point in using caches at all if coherence must be simulated in software. The example also suggests that directory schemes or other hardware coherence

mechanisms can hurt performance unnecessarily if they are automatic and transparent to the software.

In Symunix II, cacheability is visible in the user interface. A region of virtual memory may be flagged as cacheable or non-cacheable. The attribute may be changed at will, and the user can invalidate and/or flush the processor cache as needed. An analogous facility is provided by Xylem. A shortcoming in these schemes is the coarse granularity imposed by the overhead of system calls for changing virtual memory attributes, and by the inability to manipulate attributes of memory areas smaller than a page.[5]

Finer control, in both space and time, over the cacheability distinction can be obtained in Symunix II using memory region *aliasing*: that is, two different virtual address ranges, one cacheable and one non-cacheable, can be mapped to the same region of physical memory. When a variable is cacheable, the cacheable address alias is used; otherwise the non-cacheable address is used. The overhead of switching from cacheable access of a variable to non-cacheable access then involves at most a cache flush and/or invalidation.

An important application of cacheability aliasing occurs in a run-time environment for a block structured language with stack frames. If all physical memory locations have two virtual addresses in the same process, one cacheable and one non-cacheable, then adjacent variables (in the same stack frame or physical page) need not have the same cacheability attribute.[6] The compiler may generate cacheable/non-cacheable references based on what it knows about the status of the variable. Without aliasing, it is more difficult to sub-partition a stack frame according to cacheability properties.[7]

2.4. Process image considerations

The process image includes the actual text and data of a process and also certain data that store execution state information (e.g. registers). The ability to access and examine a process image, both during execution and after termination, is important to support such capabilities as interactive and post-mortem debugging and program monitoring. The need for debugging and monitoring tools is is particularly acute in the more complex parallel programming environment. Furthermore, typical applications are extremely compute intensive, and may run for days or weeks. Being able to conveniently generate restartable checkpoints of process images is thus an important goal.[8]

The standard UNIX core file does not adequately represent the post-mortem process image of parallel programs. Since a parallel program is a *set* of processes

---

[5] In Xylem, pages are flagged by the user program as *global* or *cluster*. However, in the absence of caches, the virtual memory system must transfer whole pages among layers of the memory hierarchy.

[6] However, variables with different cacheability must be isolated in separate cache lines.

[7] On the RP3, because the cache provides a partial-invalidate feature, there are actually three classes of cacheability rather than just two. Several techniques can be used to minimize the impact of the redundant mappings on the amount of useful virtual memory available in the address space.

[8] The only UNIX-based system that we know of with checkpoint/restart is Cray's UNICOS system [HK86].

that share resources, the post-mortem image must reflect the entire set of processes. If all processes in a parallel program are executing in the same current directory and exit abnormally at roughly the same time, they will overwrite each others' **core** file. At best only one of the processes will be represented; at worst, the **core** file will be garbled.

These and similar considerations have led to the following goals for the Symunix II process image representation:

- *Uniformity Of Representation.* The same representation of a process should be used by all components that need to access the process image, including such utilities as **ps**, debuggers, checkpointers, and any other status displays or tools that may be devised in the future. (The loader **a.out** format could also be merged with the uniform process image representation; this is a potential topic for future work.)

- *Independence From Kernel Storage Structures.* Reliance on **/dev/kmem** as a panacea for accessing process state is not advocated. In fact, we should have freedom to restructure, eliminate, or add kernel tables with minimal impact on the kernel interface and user mode programs. For example, elimination of the *u*-block should be possible without affecting utilities and tools.

- *Representation Of Parallel Programs.* Since a parallel program consists of multiple processes, it must be possible to reconstruct the relationships among the processes, as well as the actual data of individual processes, from information in a **core** dump or checkpoint.

- *Consistent Representation.* The issue of consistency pertains to the accurate representation of parallel programs with shared memory objects. A representation is *consistent* if it stores a valid state of computation reached during program execution. In particular, if several processes sharing memory core dump or are checkpointed by a signal, the combined saved images should represent a valid computation state. Potential problems arise because signals arrive and are handled asynchronously. If multiple copies of a shared object are made by processes receiving a signal, then copies made at different times are potentially inconsistent; even if there is only one copy of each shared object dumped, care must be taken to avoid inconsistencies between shared and non-shared objects at the time of signal handling.

  Another type of potential inconsistency in shared objects arises when sharing files across a network. If two processes on separate nodes map an address region to the same file (see below), then write operations to the individual buffer areas can result in file inconsistencies. We consider this to be a file system issue: if the network file system supports consistent read/write sharing, then that mechanism can be used to maintain consistency for shared memory segments.

- *Security.* If process images are restartable, care must be taken to prevent security holes while executing **setuid** programs.

The work described in [Kil84] demonstrates that the file system interface provides an easy and general purpose way to access the process image for debugging. We carry this idea further by actually maintaining the entire process image

as a set of ordinary files. In Symunix II, the file system serves to unify the treatment of all memory regions in an address space. There is in fact a single representation of a process image for the diverse purposes of core dumping, process monitoring, swapping/paging, and checkpoint/restart. A more detailed presentation of this design is the topic of Section 3.

3. Symunix II Memory Management

In the following, we return to all of the issues discussed in the previous sections, but in a more concrete fashion. The model of memory supported by the Symunix II kernel, as well as the set of system calls provided to manipulate memory are presented, and we contrast this design with related work.

3.1. Memory model

3.1.1. Logical and physical segments The virtual address space of a process consists of a variable number of *logical segments*, each a contiguous range of virtual addresses. The address space can be sparse; the only restrictions on placement of logical segments within the address space are that they be disjoint and that proper alignment and size restrictions for the underlying architecture be obeyed.

A logical segment is mapped into a physical memory object called a *physical segment*. If more than one logical segment is mapped into the same physical segment, the physical segment is *shared*.

Each logical segment defines a set of *virtual attributes*, which may differ among logical segments sharing the same physical segment. These include the following:

- Protection: a process can have read, write, and/or execute permission on a logical segment.
- Cacheability: a process may specify whether a logical segment is cacheable or not.
- Action on **spawn/fork**: when a process spawns or forks subprocesses, a logical segment can be (1) dropped from, (2) copied into[9], (3) inherited into (shared with), or (4) reallocated (without copying) in, the address spaces of the subprocesses.
- Action on **exec**: a logical segment can be preserved or dropped from the process image during an **exec** system call. This feature can be used to transmit environment variables in user mode, for example.

It is possible to change virtual attributes of a logical segment, or of a portion of a logical segment. (If attributes of a portion of the logical segment are changed, the segment is split into smaller segments.)

Physical segments can also have attributes. Unlike logical attributes, these *physical attributes* are specified only when the segment is created, cannot be changed subsequently, and may be machine dependent. An example on the RP3 is the physical memory type, which may be *sequential* (local) or *interleaved* (global).

[9]The copy-on-write optimization is employed (see below).

A second attribute, *strict* vs. *non-strict*, affects the interpretation of the physical memory specification. A strict allocation request will fail if the kernel is unable to obtain enough memory of the requested physical type; a non-strict request is advisory in that memory of another kind may be substituted. A non-strict request also permits eventual implementation of sophisticated mechanisms like demand paging of local pages from a larger global memory space.

3.1.2. Image files Each physical memory segment is described by an offset and a size within a corresponding *image file*. The image file, which is an ordinary file in the UNIX file system, stores the contents of the physical segment. Portions of physical segments are allocated in memory during process execution; thus main memory serves to cache image files. Memory management is thereby integrated with file I/O: every frame of main memory is, at any point in time, either (1) unused, or (2) being used to cache a block of an image file (i.e., mapped into one or more logical segments), and/or (3) being used to cache a block of a file being read or written, replacing the function of the kernel buffer pool of traditional UNIX.

Image files provide a unified framework for a set of distinct features for both parallel programming and general system performance. More specifically, mapped image files provide:

- Backing store for swapping and/or paging.
- A means for loosely related processes to share memory via the file system name space.
- A limited memory-mapped I/O facility for improving random file access.[10]
- A unified process image, consisting of a set of files storing both shared and private data. Image files enable access to the text and data segments of a process during execution, for debuggers and status display programs, and provide a permanent snapshot of the process state, for post-mortem debugging or checkpoint/restart.

There are two mechanisms provided for establishing sharing, as mentioned in Section 1.

(1) A logical segment can have its action-on-spawn attribute set to *inherit*. On spawn, the child processes will share the physical segment/image file into which the original logical segment is mapped, at the same virtual address.

(2) Two or more processes can map into the same image file via the new **mapin** system call. It is only necessary for the cooperating processes to have the appropriate permissions to access the file. Unlike sharing via inheritance, this mechanism provides sharing among processes not directly related by spawn, and it is possible for logical segments sharing the same physical segment to span different virtual ranges.

[10] In a swapping implementation, the size of a mapped file will be limited by the size of physical memory. In any case, the read system call is still needed for initializing a logical segment from a file, so read cannot be implemented purely on top of file mapping.

Physical memory segments in Symunix II are created only through file mapping. The **fork**, **spawn**, and **exec** system calls use an internal kernel version of **mapin** for this purpose. If a special file, representing a device, is mapped in, then no physical memory is allocated; instead, a logical segment is created and mapped as directed by the respective device driver.

The user program and programming environment manage the virtual address space either statically, through a generalized object file format which describes an arbitrary list of sections with respective virtual and physical attributes, or dynamically. The memory management system interface, listed below, specifies basic operations on logical segments. The first three calls modify process address space. The last two calls obtain information about the address space composition.

Our use of image files is similar in some respects to the specified but unimplemented **mmap** system call package from 4.2 BSD, which also maps files into the process address space. Variations on this package have been implemented in SunOS and Dynix, among others. Dynix, however, has a more static view of the partitioning of memory into private, shared, and stack areas than Symunix II. In the **mmap** model, file mapping is used only when memory sharing is intended. Otherwise, memory regions are allocated from *anonymous* storage, which is backed up by dedicated swap areas on disk. In Symunix II, image files serve as backing store for the entire address space; no anonymous swap area is used.

Furthermore, when an image file is mapped to several read/write logical segments in Symunix II, it is fully shared by all of them; writes by each are visible to all. If it is desired that changes remain private, then it is necessary that the

mapin(*vaddr*, *size*, *imagefd*, *imageoff*, *flags*)	Create a physical segment from the image file accessed through *imagefd* at file offset *imageoff*; map it into the virtual segment beginning at *vaddr*. Virtual and physical attributes are given by *flags*. (To initialize from another file, use **read**.)
mapout(*vaddr*, *size*)	Unmap the virtual address range beginning at *vaddr*. (*Vaddr* need not begin a previously allocated logical segment.)
mapcntl(*vaddr*, *size*, *flags*)	Change the virtual attributes of the address range beginning at *vaddr* according to *flags*.
pinfo(*pid*, *parea*)	Obtain basic (fixed size) information about the process image of process *pid* returned in location *parea*.
pmemaddr(*pid*, *parea*, *vaddr*)	Obtain information about the logical segment that includes *vaddr* for process *pid*.
pmemvect(*pid*, *parea*, *psize*)	Obtain a vector of information, of *psize* bytes, about all logical segments in the address space of process *pid*.

Table 1.

processes create private memory segments (i.e., mapin separate image files).[11] We do not provide an analog to the MAP_PRIVATE/MAP_SHARED option of **mmap** as in SunOS, through which some of the sharing processes may request that their writes be propagated to private copies in anonymous storage, while other processes sharing the same pages request that their writes be reflected in the original, shared copy. We also do not provide an **msync** system call to flush modified pages in memory to disk, as in SunOS, since the existing **fsync** primitive satisfies this requirement at the kernel interface level.

3.1.3. User mode address space management The Symunix II kernel does not interpret particular virtual memory ranges in any special way. Virtual address space management is entirely the responsibility of the user mode layer. This decision was made both to avoid duplication of function and because allocated memory may be shared in complex patterns among many processes. It is undesirable to establish policies in the kernel for coordinating address space layouts among sets of processes.

Thus, while the kernel creates mappings between process address spaces and image files (and thus physical memory), it is left to user programs, compilers, linkage editors, or libraries to establish sharing of memory. In particular, a C library interface, extending **malloc** and **free**, is provided for dynamically allocating private and shared memory that:

- hides explicit references to image files and other lower level system interface details from the user;
- manages the address space of a process or, more importantly, of a family of processes, so that shared memory regions can be allocated at identical virtual addresses within the processes of the family;
- implements sharing, allowing a user to issue a single request to allocate or free a region of shared memory, even after spawns;
- allows for setting up virtual aliases of regions, for specifying virtual and physical attributes, and for mapping regions into files.

The mechanics of establishing sharing are hidden through a special SIGSEGV handler, which detects when it is necessary to mapin a shared region for the current process. (Dynix uses a similar technique.) Furthermore, since "automatic" stack growth must be handled in user mode, the same package, again using the SIGSEGV handler, allocates memory for stack growth according to the stack discipline in use.

3.2. Management of the process image

During execution, the text and data of the process image are contained in a set of image files, which can be accessed subject to the file permissions. According to local installation policy, image files may be kept in a special directory hierarchy or

[11] Nonetheless, copy-on-write may be used among the private segments, subject to alignment requirements of the underlying hardware, when they are initialized from a common file through **exec** or **read** system calls.

file system. *Reference directories*, described below, may be used for managing administrative conventions in user mode.

3.2.1. Reference directories We introduce the notion of *reference directories* largely for performance reasons, but they play a role in the memory management interface as well. Image files are created implicitly by the kernel on **spawn**, **fork**, and **exec**; we need to be able to specify a "current" image file directory, analogous to the traditional current working directory, where these image files will be stored.

Path search, i.e., finding an inode from a file name, is traditionally an expensive process in UNIX, depending on the depth of the directory hierarchy that must be traversed.[12] This search is eliminated for file references relative to the current working directory by keeping the inode pointer for that directory in the *u*-block.

Reference directories extend this feature. Any frequently accessed directory may be designated as a reference directory and bound to a *refname*; its inode is then saved to avoid further path searches. *Refnames* are prefixed with the character '@'. The *refname* **@image** is specially reserved by the kernel to designate the repository for the current process' image files, in particular, for those created implicitly by the kernel. Reference directories are defined and removed via the functions:

mkrefdir(*path*, *refname*)
fmkrefdir(*fd*, *refname*)
rmrefdir(*refname*).

Users can call **mkrefdir** to change the **@image** directory, or to define other frequently referenced directories (e.g., directories in $PATH). Files in the reference directory may be accessed with the name *@refname/filename*.

3.2.2. Core Files An important question is the dispensation of the image files upon process termination. When a file is created solely to act as an image file, it is typically of interest only as long as the processes mapping the file are still running. To avoid cluttering up the file system, such a file should disappear when the last process that maps it terminates. However, there are sometimes good reasons not to delete image files upon process termination. In the event of abnormal termination, image files comprise the process data needed for post-mortem analysis and possible restart.

To allow for user control over the fate of image files, Symunix II defines additional flag options on the **open** and **fcntl** system calls. These options specify action on final close operation: always delete the file on the last close operation; delete the file *unless* a process that has opened it terminates abnormally; and delete the file if a process that has opened it terminates abnormally. (This feature implies that file names of opened files are remembered by the system.)

When a core dumping signal is received, the kernel creates a file called **core.***pid* that stores the process state information. (As usual, *pid* is the process identifier of the current process.) The information stored should be sufficient for the process to be restarted. The process image text and data are already available

[12] In 4.3 BSD, the path search overhead is reduced through use of a name translation cache.

in the image files for the process. Moreover, upon restart, shared image files will be consistent because they store the most recent updates to shared variables at the time the last process sharing the files terminated.

3.3. Checkpoint/restart

A checkpoint/restart facility is planned in Symunix II through a new SIGCHKPT signal and a **restart** system call. A core dump corresponds to a *terminal checkpoint*; other checkpoints (triggered by SIGCHKPT) require copying the relevant set of image files and the core.*pid* file. The **restart** system call, essentially a generalized exec, restarts a process, family, or process group from a set of process images. Each checkpoint may be generated either on request by an executing program, or at regular intervals defined by the program or an external agent. An executing program can suppress or intercept scheduled checkpoints. (In a garbage-collected language environment, for example, the runtime system could arrange to compact storage before checkpointing.) Several mechanisms for control of periodic checkpointing are under consideration; this is a topic of ongoing work.

Unlike terminal checkpoints, the checkpointing of continually running process aggregates (families or process groups) requires synchronization to ensure consistency. In order to obtain a consistent snapshot of an executing multi-process program with shared memory, it is necessary to suspend execution of all processes of the job, copy the image files, and then resume the processes.

We would like to provide a checkpoint/restart facility which is useful in a general setting including interactive parallel applications and utilities such as editors (for recovery), debuggers, and conceivably login sessions (shells), as well as long-running computational "batch" jobs. Because disk I/O will be the critical expense, and because process memory will often remain largely unchanged between checkpoints, we plan to investigate the feasibility of implementing copy-on-write in the file system to reduce redundant disk writes and speed up checkpoints (and other file operations).

4. Performance and Implementation Issues

The memory management model presented raises a number of implementation issues, several of which are considered in this section. Crucial efficiency questions introduced by the use of image files are addressed, after which aspects of the implementation of logical and physical segments are discussed very briefly.

4.1. Performance of memory allocation with image files

Several clear performance benefits derive from the above-described approach to memory management. Because file I/O buffers and process memory regions compete for physical memory resources, the system can adapt to dynamic requirements, in particular to maintain effective file cacheing. Memory-mapped I/O can in many cases speed up file read and write operations, by eliminating buffer copying while retaining the buffer cacheing function.

However, we have described a system in which every request for allocation of process memory must be preceded by a file creation. This would seem to impose unacceptable overhead. Furthermore, image files are generally very transient, and it is undesirable to clutter the file system with many of these temporary files. In

order to see how these problems can be avoided, we must first consider exactly what factors contribute to the expense of file creation and deletion operations.[13] The major suboperations are:

- Path searching to find the inode for, and open if necessary, the parent directory for the file to be created.
- Reading the entire directory to check whether the file already exists.
- Allocation and initialization of an in-core inode.
- Allocation of disk inode and writing inode to disk.
- Updating directory with new file name.
- For delete: freeing of any allocated data and indirect blocks, removal of disk inode, removal of file name from directory.

The use of reference directories for image files eliminates the path search overhead. Further optimizations stem from two observations concerning image files:

(1) There is no need for a physical segment (or image file) to have a name if there is no reference to that name. This will generally be true of private or inherited segments, and is often true of dynamically-shared segments because sharing processes can in many cases **mapin** an image file by importing an open file descriptor.[14]

(2) If the physical segment is never flushed to disk, because no pageouts, swapouts, or checkpoints occur before the segment/image file is deleted, then there is no need for the existence of the image file to be registered on the disk at all.

These are exploited using a strategy of *lazy file creation*. When a file is newly created in this mode, no inode need be written to disk until the first actual I/O operation, and no name need be created in the directory until it is needed for a subsequent open or until the directory is itself read (e.g., via **ls**).[15]

Lazy file creation is requested with a new flag option in **open**, which causes files to be created *anonymously* in a specified directory. The meaning of the anonymous option is that a new filename is generated by the system; the path argument to **open** specifies a directory only. The following call creates an anonymous file in the reference directory *@image*, though any directory name could have been used.

```
open("@image", O_CREAT | O_ANON)
```

The file descriptor of the anonymous file is returned. For most references to the

[13] In some swapping implementations, the lack of contiguity of data blocks in the file system further increases the expense of swaps. This can be addressed using a file system which supports contiguous pre-allocation of files or contiguous extents. In the current section we assume an architecture in which page frames are normally discontiguous in memory.

[14] This is made possible through a new system call, **pdup**, which *dup*'s open file descriptors across processes running under the same userid.

[15] Since it is possible that a directory entry will be created for a file before its inode is written to disk, the **fsck** utility must be able to properly handle this situation.

open file (e.g., **mapin**), this is sufficient. A file name need not be generated and installed by the system until the *@image* directory is read to the end or the name is explicitly requested by the call,

fdname(*fd*, *fname*, *size*).

For this reason we prohibit **mapin** on directories.

Creation of the incore inode remains as the only operation in the above list which must *always* be performed when a file is created for **mapin**. In Symunix II, a mapped-in inode is, in fact, the primary locus of reference and control for a physical memory segment. The internal organization of the memory management component is described briefly in Section 4.4. As of this writing, no data exists for evaluating the overall performance ramifications of this system.

4.2. TLB coherence and demand paging

One approach for solving the TLB consistency problem on multiprocessors (see [TKS88], [RTY87]) is "TLB shootdown", in which a processor making certain non-local mapping changes must broadcast a change to other affected processors and wait for them to update their TLBs and respond before proceeding. Because of the potential expense of this mechanism, especially on large multiprocessors, Symunix II avoids non-local mapping changes entirely. Each operation specified in the above interface affects only the current process. Global mapping changes are accomplished in user mode, where synchronization can usually be avoided.

This policy suggests a somewhat conservative approach to paging. Pages are not evicted while actively mapped into multiple concurrent processes' address spaces. The current (very conservative) implementation is a hybrid of swapping and paging, in which no process can run until sufficient memory is available for the entire process image. However, page frames are not actually allocated or initialized until referenced. On swapout, pages are placed on an available list from which they may be reclaimed by the original process or allocated (and written out to the proper image file if modified) by another process.

4.3. Copy-on-write

Copy-on-write is not a form of memory sharing, but rather an optimization useful on some architectures when logically private copies of memory areas are created. It is invoked in Symunix II in three circumstances, provided that architecturally-imposed size and alignment restrictions are satisfied:

(1) On **fork** or **spawn**, when a logical segment is marked to be copied to child processes.

(2) On **exec**. By creating text and initialized data segments as copy-on-write copies of the *a.out* file, we retain the advantages of shared text when the text is read-only, and resolve two traditional drawbacks to shared text. Because any text page can be privately modified without affecting other sharers, breakpoints can be implanted by a debugger, and the disk file can be rewritten, while the file is shared. Thus **ptrace** becomes more flexible, and the notorious ETXTBSY error (can't open file for output because it is a shared text program being executed) is eliminated.

(3) On **read**. When an image file is read with the **read** system call and the relevant page is resident, or when multiple processes read the same block of a file, the copy-on-write optimization is appropriate.

Our TLB coherence policy will occasionally limit the use of copy-on-write, in particular on **read** operations. If any region involved in a copy is already shared, then it is impossible to set the pages to read-only status (so as to catch writes) without non-local TLB changes.

4.4. Logical and physical segment representation

Each process has an *address space table*, which contains a list of *logical segment descriptors* (*lsegs*) for the process. Each lseg stores the virtual address and size of a logical segment, virtual attribute information, a pointer to the inode of the image file representing the physical memory segment, and the offset into the image file. The inode contains a count of referencing lsegs. Buffer headers for memory-resident pages of the image file are accessible directly from the inode, indexed by file offset. The buffer headers perform their conventional role with respect to I/O, and at the same time represent pages of the corresponding physical segment. Each buffer header contains a pointer to a page frame which holds the buffer data; in the presence of copy-on-write several buffer headers may share a single frame.

The machine dependent layer manages page tables or other structures needed for a particular architecture. On some machines, it is important that mapping structures (e.g. segment and page tables) be shared in the common case that multiple lsegs share physical memory segments at identical virtual addresses. Ideally, when two processes have identical virtual-to-physical mappings with identical attributes, the mapping structures should be entirely shared. One scheme, currently under investigation, involves associating the machine-level memory management structures with their corresponding logical segments through a hashing mechanism. The hash function is designed to accomplish the sharing of mapping structures whenever possible.

5. Status

Symunix I has been running on an 8 processor Ultracomputer prototype since summer, 1984. A preliminary version of Symunix II, with the new memory management interface but without image files, has been debugged on the Ultracomputer prototype, and on a single processor RP3 simulator. (The current system is symmetric, but not highly parallel because it relies on very large critical sections.) Development of the remainder of the system is in progress.

6. Acknowledgements

The ideas described in this paper have benefited from many discussions with other members of the Ultracomputer Research Laboratory at NYU and, also with researchers at the IBM RP3 project. Eric Freudenthal contributed to the design of the high-level memory management library, and Vicki Myroni is porting the system to the IBM RT PC. We also thank Wayne Berke for careful reading of the paper.

References

[AB85] Archibald, J., Baer, J., "An Economical Solution to the Cache Coherence Problem," *Proc. 12th International Symposium on Computer Architecture*, 1985.

[BW88] Barton, J., Wagner, J., "Beyond Threads: Resource Sharing in UNIX," *Winter USENIX Conference Proceedings*, 1988.

[Ba78] Baudet, G. M., "Asynchronous Iterative Methods for Multiprocessors," *Journal of the ACM*, Vol. 25, No. 2, pp. 226-244, April 1978.

[BO87] Beck, B., Olien, D., "A Parallel Process Model," *Winter USENIX Conference Proceedings*, 1987, pp. 83-101.

[CF78] Censier, L., Feautrier, P., "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, c-27(12):1112-1118, 1978.

[EGL86] Edler, J., Gottlieb, A., Lipkis, J., "Considerations for Massively Parallel UNIX Systems on the NYU Ultracomputer and IBM RP3," *Winter USENIX Conference Proceedings*, 1986.

[ELS88] Edler, J., Lipkis, J., Schonberg, E., "Process Management for Highly Parallel UNIX Systems," *Proceedings of USENIX Workshop on UNIX and Supercomputers*, September, 1988.

[GMS87] Gingell, R., Moran, J., Shannon, W., "Virtual Memory Architecture in SunOS," *USENIX Proceedings*, June 1987, pp. 81-94.

[GLR83] Gottlieb, A., Lubachevsky, B., Rudolph, L., "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 2, 1983, pp. 164-189.

[Got87] Gottlieb, A., "An Overview of the NYU Ultracomputer Project," *Experimental Parallel Computing Architectures*, ed. J.J. Dongarra, North Holland, 1988, pp. 25-95.

[HK86] Hoel, T,. Keller, B., "A Unix-based Operating System for the Cray 2," *Winter USENIX Conference Proceedings*, 1986, pp. 219-224.

[Kil84] Killian, T.J., "Processes as Files," *USENIX Summer Proceedings*, 1984, pp. 203-207.

[ME87] McGrath, R., Emrath, P., "Using Memory in the Cedar System," *Proceedings of the First International Conference on Supercomputing*, Athens, Greece, June 1987. *Lecture Notes in Computer Science*, Vol. 297, Springer-Verlag (Berlin/Heidelberg), 1988, pp. 43-67.

[OCD88] Ousterhout, J.K., Cherenson, A.R., Douglis, F., Nelson, M.N., Welch, B.B., "The Sprite Network Operation System," *IEEE Computer*, Feb., 1988, pp. 23-35.

[PBG85] Pfister, G., Brantley, W., George, D., Harvey, S., Kleinfelder, W., McAuliffe, K., Melton, E., Norton, V., Weiss, J., "The IBM Research Parallel Processor Prototype: (RP3): Introduction and Architecture," *Proceedings of*

the 1985 International Conference on Parallel Processing, Aug. 1985, pp. 764-771.

[RTY87] Rashid, R., Tevanian, A., Young, M., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1987, pp. 31-39.

[TG88] Teixeira, T., Gurwitz, R., "Stellix: UNIX for a Graphics Supercomputer," *Summer USENIX Conference Proceedings*, 1988.

[TKS88] Teller, P., Kenner, R., Snir, M., "TLB Consistency On Highly-Parallel Shared-Memory Multiprocessors", *Hawaii International Conference on System Sciences*, 1988.

[TR87] Tevanian, A., Rashid, R., Young, W., Golub, D., Thompson, M., Bolosky, W., Sanzi, R., "A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach," *USENIX Proceedings*, June 1987, pp. 53-67.

Reduction of Static and Dynamic Memory Requirements on the Cray X-MP

E. C. Pariser

AT&T Bell Laboratories

ABSTRACT

Many large scale computing applications are not only compute-intensive, but memory intensive as well. On a virtual memory machine a large job that requires significant paging will run considerably slower than one that does not, while on the Cray X-MP, which is not a virtual memory machine, central memory size is an even more significant limitation. For these large jobs reduction of memory requirements are necessary to enable a job to run at all on the machine, and therefore take precedence over vectorization and optimization questions. As well as some programming structure techniques for both C and FORTRAN, this paper discusses two methods for reducing the size of the user job, one specific to the Cray X-MP, and one applicable to any UNIX® system. The first method is to access the Cray Solid State Disk, which has a transfer rate of 1250 Megabytes/second. The second method is to set up and use an efficient dynamic memory management package with garbage collection based on UNIX system routine primitives.

1. Introduction

The class of supercomputers, because of their speed, are the main support of large, compute intensive simulations and applications. Many of these applications require correspondingly large amounts of central memory and working storage, and are incapable of running on a machine which does not meet these requirements. The Cray X-MP is a physical memory machine, limited to 32 or 64 Megabytes of central memory (4 or 8 64-bit integers). As the UNIX® operating system on this machine uses approximately 6.4 Megabytes of central memory, this strains even further the maximum memory limit that a job can reach. The need for ways to reduce the memory requirements of a running job are often of primary importance to the application programmer because large, rigid memory requirements may completely eliminate the Cray X-MP as a possible location for the application.

Even when an application will fit and will run on the Cray, on time-sharing systems there are secondary advantages to keeping the memory requirements small. On the UNIX time-sharing system, a running user process resides in a single contiguous block of memory. In order to start, the user process must wait for a large enough contiguous block of central memory to become available. It is also possible that a user job might be moved out of central memory to a swap device to make room for another job that is ready to run. Not only do many job schedulers place large jobs higher on the candidate list for being swapped out, but having been swapped out, once again the appropriate amount of space must be made available in

central memory before the large job can be swapped back in. Smaller jobs will thus often have a shorter wait time on the swap device than larger jobs, and the jobs themselves take less time to transfer back than larger ones, simply by virtue of the size of the data transfer.

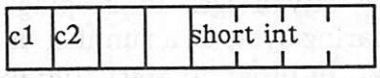
This paper first briefly discusses some techniques for reducing the memory requirements by changing the coding structure of the program, then discusses in greater detail two more complex methods for reducing the memory requirements of a job running on the Cray-X/MP under the UNIX operating system. The first method is the use of the Cray X-MP Solid State Disk (SSD). Timing results are given for different methods of accessing the SSD, and techniques for altering code to reduce the static memory requirements of a job and to run efficiently to this device are also discussed. The second method discusses the use of a dynamic memory management package with garbage collection based on UNIX system calls. A short explanation of dynamic memory management is presented, with suggestions for coding style. Additionally, the problems peculiar to the UNIX system on the Cray X-MP that were encountered while designing and implementing one such package are discussed.

2. Restructuring Code

There are many aspects of code that need to be looked at when moving code to memory limited machine. Some of these methods are quite simple and straightforward to implement, some are less straightforward or simple. They are, however, the first step in changing the code to allow for a physical memory machine, as cleaning up code is always a good first step.

Overlapping Storage: After finding all the unused variables in the code (this step will not be necessary if you wrote the code yourself, but is always necessary when inheriting code from others), the most straightforward area to address is the storage given up as temporary work areas. One way to deal with these is to allocate the storage needed as common blocks, so that maximum overlap of temporary areas occurs. In a similar vein, on a smaller scale, equivalence or union structures can be used to allow for overlap of individual data structures.

On the Cray, such statements must be used with care, particularly when character variables are involved. The Cray word is 8 bytes long and is packed with characters from left to right. Thus overlaps do not occur where they would on other machines. For example, the union or equivalence of a short with two characters would have no overlap at all, as shown in the figure below. If the value of the short is being tested as a fast way to test the value of the two characters simultaneously, on the Cray this test will have no validity.



When porting C code, a more subtle problem to unravel is the union of an integer with a character pointer. The Cray uses a segmented addressing scheme, it maintains addresses as a word address plus a byte offset into that word. This byte offset is held in the three high-order bytes. Integers, in general, do not have this

format. Thus calculating the distance in bytes from location zero or another known base address yields an integer number of bytes which cannot be correctly used as an address. This integer can be used in pointer arithmetic, as the appropriate transformation and scaling is performed by the compiler, but the integer cannot be used directly as an address, as illustrated in the examples below.

```
        int i;
        char d1, d2;

/*  these two statements will correctly calculate value  */
        i = &d1 - &d2;
        if (i % 8) {    /* testing for non-integral # of words */

/*  this test will not correctly determine a word boundary  */
/*  because the byte offset is stored in the high order bytes of d2  */
        if (d2 % 8) {   /* testing for word boundary */
```

Before use as an address, an integer must be explicitly transformed to a pointer; the same is true when using a pointer to represent an integer byte count. The subroutines below will accomplish these tasks.

```
/* convert a pointer to a character address int */
unsigned ptr2add( ptr )
        unsigned ptr;

        {       unsigned waddr, char_offset;
                char_offset = ( ptr >> 61 ) & 0x7;
                waddr = ptr & (unsigned)0xffffffff;
                return( (long)((waddr << 3) | char_offset) );
        }

/* convert a character address int to a pointer */
char *add2ptr( addr )
        unsigned addr;

        {       unsigned waddr, char_offset;
                char_offset = addr & 0x7;
                waddr = addr >> 3;
                return( (char *)((char_offset << 61) | waddr ) );
        }
```

Condensing storage: When writing code for a vector machine such as the Cray X-MP, it is often necessary to trade temporary storage space for vector speed. When done on the base of syntax alone, a memory intensive code might be prohibited from vectorizing because of lack of space. Examing the code with knowledge of data flow through the program, and writing or rewriting code to condense storage as much as possible often yields large gains. In the example below, if the programmer knows that the conditional fails significantly more often or less often than 50% of the time, recoding the section will return considerable storage space. The code given assumes a failure rate much greater than 50%; analogous code could be generated for a failure rate much less than 50%. Of course, other code sections

will have other constraints. Note that the temporary storage area X is in use to allow the original loop to vectorize.

```
C  THE ORIGINAL CODE; IT WILL NOT VECTORIZE BECAUSE OF CALL TO MYSUB
        DO 10 I = 1,N
          IF (B(I) .NE. 0)
          THEN
            B(I) = MYSUB (B(I)) * SIN (OMEGA*B(I))
          ELSE
            B(I) = 3.14
          ENDIF
10      CONTINUE

C  CODE CHANGED TO ALLOW VECTORIZATION OVER PART OF THE CODE,
C  STORAGE INCREASED
        DO 10 I = 1,N
          IF (B(I) .NE. 0) X(I) = MYSUB (B(I))
10      CONTINUE
        DO 20 I = 1,N
          IF (B(I) .NE. 0)
          THEN
            B(I) = X(I) * SIN (OMEGA*B(I))
          ELSE
            B(I) = 3.14
          ENDIF
20      CONTINUE

C  CODE CHANGED TO REDUCE STORAGE IF B(I) IS GENERALLY ZERO
        K = 0
        DO 10 I = 1,N
          IF (B(I) .NE. 0)
          THEN
            K = K + 1
            INDX(K) = I; X(K) = MYSUB (B(I))
          ELSE
            B(I) = 3.14
          ENDIF
10      CONTINUE
        DO 20 I = 1,K
          B(INDX(I)) = X(I) * SIN (OMEGA*B(I))
20      CONTINUE
```

Choice of Algorithm: There are often sections of code that can be replaced entirely by a more space efficient algorithm. While this often results in a tradeoff of space for time, in some cases it does not. Most notably, moving from a quicksort variant to the old in-place bubble sort not only uses less storage but runs faster as well. First, the quicksort uses a working storage vector while an in-place sort does not. Second, the standard bubble sort, on a vector machine, vectorizes extremely well and maintains long vector lengths, which is something to shoot for as it means lower overhead. The quicksort does not vectorize as well, there is more

scalar setup and overhead, and the vector lengths get progressively shorter as the algorithm proceeds. A good comparison of sorting routines on supercomputers including the Cray X-MP can be found in Reference [1].

3. Static Memory Size: Using the SSD

The SSD is a MOS memory device configured under the UNICOS operating system as disk storage. As it is essentially dynamic random access memory, it has a much lower seek time than disk by a factor of 4000 (4 microseconds vs. 16 milliseconds). Writing code to access the SSD instead of creating large data storage areas can significantly reduce memory requirements without noticeably slowing down the execution time. In fact, using this device can eliminate much of the system overhead and wait time associated with large jobs without adding much additional data transfer time, possibly reducing the elapsed time of the running job by a tremendous amount. In one case, this reduction amounted to nearly 50% of the running time of the code, while reducing the storage requirements by 40%.

The channel between the SSD and central memory is highly parallel and has a maximum theoretical transfer rate of 1250 Megabytes per second. In actual tests, as shown below, the maximum rate achieved by directly accessing the device through system calls is nearly as good as theoretical, over 1000 Megabytes per second. In contrast, the speed of the Cray DD-49 disk drive is theoretically 9 Megabytes per second with an average speed of about half that; that is, the data transfer rate between central memory and the SSD is 200 times faster than between central memory and disk. It is possible to access the SSD as a standard disk through the standard FORTRAN (or C) I/O libraries; however the overhead of these libraries overwhelms the speed of the hardware. As can be seen from the table, the overhead involved in using the disk directly through system calls is small enough that it quickly becomes negligible. The numbers in the third column for FORTRAN access through the standard libraries are for unformatted I/O, as formatted I/O is of necessity much slower (Ref. [2]).

Cray X-MP SSD Performance Data Transfer Rate (Megabytes/second)				
Number of SSD Blocks	Number of Bytes	I/O Library Disk	I/O Library SSD	System call to SSD
2	8 K	4.1	8.7	402
8	32 K	4.4	9.2	775
32	128 K	4.5	9.3	996
128	512 K	4.7	9.3	1079
256	1024 K	4.9	9.3	1091
400	1600 K	5.4	9.3	1096

These tests were made on a relatively quiet system, although it is recognized that many factors affect the performance of any transfer routine used. These measurements were made using the system real-time clock, which has 9.5 nanosecond accuracy. The I/O operations were repeated several times, so that the overhead of accessing the system clock did not affect the significant accuracy. In

the case of the fastest routines, this required that the block transfer operations be repeated 1000 times for each measurement.

Direct access to the SSD through system calls from both FORTRAN code and C code is actually performed through three simple front end routines: *ssbrk*, *ssrd*, and *sswrt*. *Ssbrk* increases the size of the user allocation on the SSD. This allocation is addressed in blocks, starting with block 0 for each user job. *Ssrd* and *sswrt* perform simple data transfers, by blocks of 4K bytes. These routines are addressable from either C or FORTRAN (the suffix C or F to the routine name identifies the C or FORTRAN version, respectively). The argument forms vary slightly between the two languages, and the C routines return a value that is passed as an additional argument in the FORTRAN routines, but functionally they are the same.

The allocation routine takes two arguments: the number of elements requested and the size (in 8 byte words) of an element. The values returned are: the number of words allocated, the number of (4096 byte) SSD blocks allocated, and the block address of the first block of this allocation. The read/write routines take the starting address in central memory to access, the starting block number on the SSD to access, the length of an element (in 8 byte words) and the number of elements to be transferred. All these routines return the number of words allocated, -1 on error.

One additional argument was found to be necessary for the read/write routines: a flag identifying whether the transfer, if not a multiple of 4096 bytes (not an integral number of SSD blocks) should be padded, truncated or buffered to the exact size of the request. Buffering the transfer adds considerable overhead and can cut the transfer rate by nearly 50%, as it involves handling the last block of the request as a separate transfer. It may be unavoidable, however, when reading from or writing to the array closest to the high end of memory to ensure that no memory fault occurs, or when writing to the end of an array of exact size.

Use of these routines is only slightly more complex than putting the routines in place, as the following examples show.

FORTRAN Example

```
C 10000 DATA ELEMENTS ARE WRITTEN OUT TO THE SOLID STATE DISK,
C THEN THE DATA IS RETURNED TO CENTRAL MEMORY ONE BLOCK (4096 BYTES)
C AT A TIME.

      DIMENSION A(60000), B(600), SUM(100)
      INTEGER RVAL, BASE, RSIZE, TYPE, BUF
      TYPE = 1
C ALLOCATE ENOUGH STORAGE FOR A COPY OF ARRAY A,
C STOP IF AN ERROR OCCURS
      CALL SSBRKF( TYPE, 60000, RSIZE, NBLKS, BASE )
      IF (RSIZE .LT. 0) STOP
C INITIALIZE A
      DO 1 I = 1,60000
        A(I) = FLOAT(I)
 1    CONTINUE
```

```
C  COPY A TO THE SSD
C  BUF = 1 MEANS PAD THE REQUEST, I.E. COPY GARBAGE AT ARRAY END
        BUF = 1
        CALL SSWRTF( A(1), BASE, TYPE, 60000, BUF, RVAL )
        IF (RVAL .LT. 0) STOP
C  READ VALUES FROM THE SSD TO ARRAY B, 600 AT A TIME
C  BUF = 2 MEANS BUFFER THE TRANSFER TO EXACT SIZE
        BUF = 2
        DO 3 I = 1,100
          CALL SSRDF( BASE, B(1), TYPE, 600, BUF, RVAL )
          IF (RVAL .LT. 0) STOP
        DO 4 J = 1,600
          SUM(I) = SUM(I) + B(J)
4       CONTINUE
3     CONTINUE
        STOP
        END
```

Example 2: C

```
/*  This program performs the same operations as the preceding example.
    The constants sent to malloc are: 512 words/block and 8 bytes/word.
main()
{
        float a[6000], b[600], sum[100];
        int  base, rsize, i, j, type, buf;
/*  calculate length of element of a    */
        type = sizeof(float)/sizeof(int);
/*  allocate space on the SSD.  Return value is number of words read or -1  */
        rsize =  ssbrkc( type, 60000, &nblks, &base );
        if (rsize < 0) exit(rsize);
/*  initialize a and copy to the SSD.  buf=1 means pad the request   */
        for (i=0; i<60000; i++)
                        a[i] = (float)i;
        buf = 1;
        rsize = sswrtc( a[0], base, type, 60000, buf );
        if (rsize < 0) exit(rsize);
/*  copy data out of the SSD into array b.
    buf=2 means buffer the last block of the transfer   */
        buf=2;
        for ( i=0; i< 100; i++ )
        {
                        rsize = ssrdc( base, &b[0], type, 600, buf );
                        if (rsize < 0) exit(rsize);
                        for (j = 0; j < 600; j++)
                        sum[i] = sum[i] + b[j];
        }
}
```

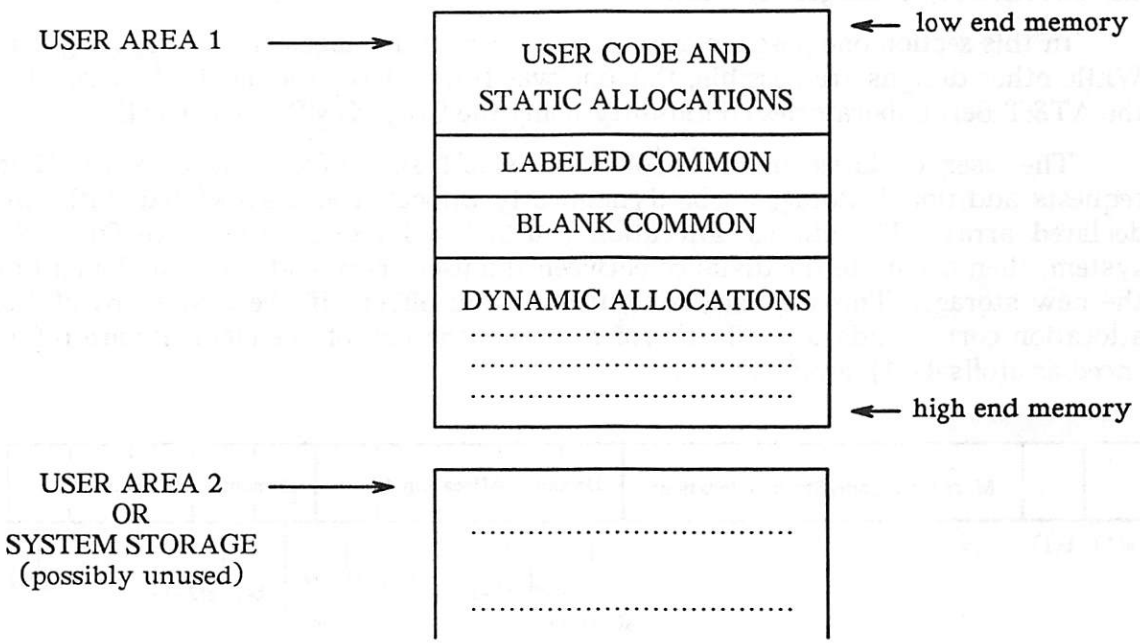
4. Dynamic Memory Management

4.1 Why Use Dynamic Memory Allocations

Often the amount of memory required by a particular program varies greatly from one run to another, depending on the specific data being analyzed. When the maximum memory requirements are large or vary greatly during the running of the program, minimizing job size by using dynamic allocation of memory is desirable. With static allocations of memory, the programmer must write code that transfers the appropriate arrays in and out of central memory, regardless of how much of the array is actually used. If the storage requirements are large for only a small number of runs, or if the requirements are large only for a small portion of the code, dynamic allocation may be more effective than use of a secondary storage device such as the SSD.

With the introduction of dynamic allocation of memory come two other necessary actions: deallocation of memory and garbage collection. In the running of a process, it is often necessary to create and access data as an intermediate step, after which the data is no longer needed and could be returned to the system. If the user deallocates a block of memory that lies between the compiled instruction code and a second dynamically allocated block, this block cannot actually be returned to the system for system use, as this would break the job into two non-contiguous areas. Thus allocated areas and free areas become mixed (memory fragmentation). After several allocations and deallocations of varying sizes, a request for one large block of memory may fail even though there is actually enough free space to accommodate the request. In this case it would be desirable to coalesce the memory allocations, or *garbage collect*. Once all allocated space is contiguous and sits adjacent to the compiled program code, (in this paper called the "low end" of memory) as shown in the figure below, the now contiguous free areas at the high end of memory can either be returned to the system to reduce the size of the job or can be reallocated, as required by the program.

The UNIX system standard C library provides some functions that perform simple allocation and deallocation of memory (see **malloc(3C)**, ([3], [4]). There are two problems with using these routines directly. First, these routines are low level and give the user little more than raw chunks of memory with no easy way to garbage collect, to examine validity of allocation areas, or to map current allocations and fragmentation. Second, these routines are C routines that are not directly callable from FORTRAN, leaving the FORTRAN programmer with the task of writing a calling interface anyway. A solid user front end can significantly ease the work of the programmer in setting up dynamic memory allocation code.



4.2 How Dynamic Memory Management Works

In general, when a programmer[1] declares a static array, i.e.

```
dimension a(10)
```

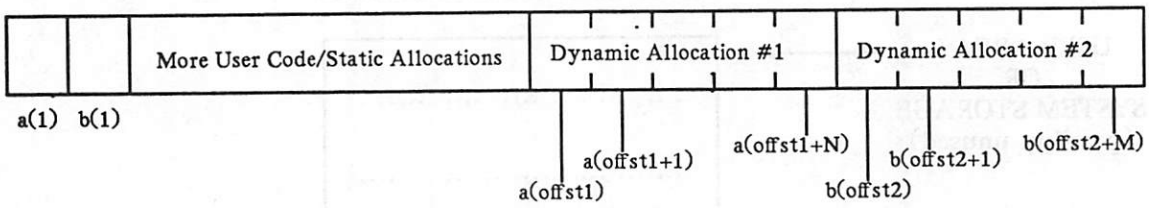
the programmer generally makes every effort to access only valid elements of this array. Most compilers do not verify the validity of the subscript. It adds considerable overhead time to the running of the program, although most can perform this activity if requested. If out of bounds checking is not in effect, then an erroneous array reference will be processed as if it were valid. A reference to a(12), for example, will access the word in memory that lies two words beyond a(10) regardless of the size of the actual declaration. If the element lies outside the user area a memory fault will be generated; otherwise, whatever value lies in that word will be retrieved. It is precisely this overlapping of memory elements that allows the elements of a dynamic memory allocation to be used. If the array is of a type that has double length elements (double precision or complex variables), the compiler handles this by scaling each subscript, as it does when valid memory locations are concerned.

1. The code examples are applicable to C as well as FORTRAN, although they are presented in FORTRAN syntax.

4.3 The Memory Management Package

In this section one possible design for a memory management package is given. While other designs are possible, this one was felt to have the most advantage for the AT&T Bell Laboratories community using the Cray X-MP (see [5], [6]).

The user declares an array of some small size, often one element, then requests additional storage to be dynamically allocated and associated with this declared array. The storage allocation routine will request the space from the system, then calculate the distance between the user array and the first element of the new storage. This distance is returned as an offset. If the first word of the allocation corresponds to a(offset), for example, the rest of the elements are referenced as a(offset+1), a(offset+2), etc.

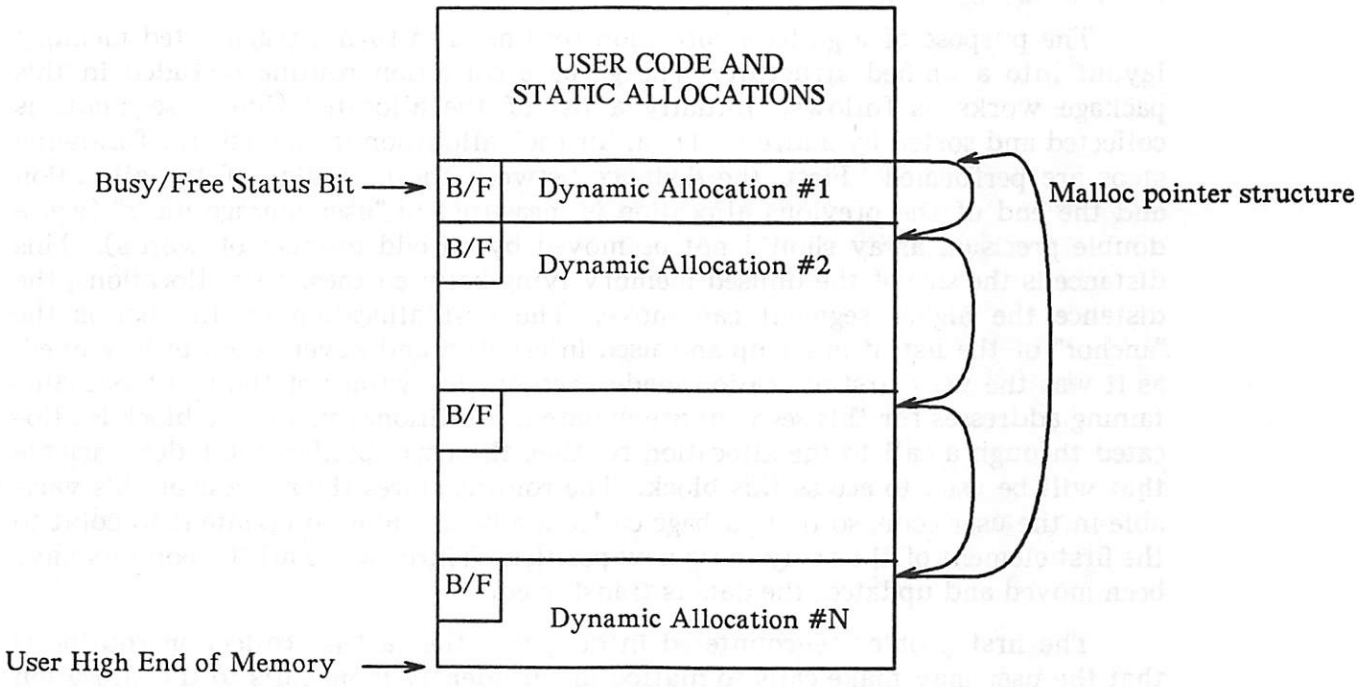


The memory allocation routine takes as arguments an array name, a desired number of elements, and return the offset from the array element to the first element of the allocation.

When the first block of storage is requested from the system through a call to **malloc**, a circular linked list is set up to keep track of areas in use. Each subsequent request is padded by one or more words which will be used to store list pointers. Associated with each block is a bit identifying the status of the block. When storage is allocated this bit is set to BUSY; when the user deallocates an area (through a call to **free**) this bit is set to FREE. **Malloc** actually searches for a FREE block large enough to accommodate the allocation request before asking for more memory from the system. Only FREE areas immediately below the user high end boundary are eligible to be returned to the system to shrink the overall user area.

Malloc and **free** are difficult to call from a FORTRAN program for two reasons. First, they are C language functions, so an interface must be written to capture the returned location of the allocation. Second, the value returned is a word address relative to the beginning of the user area, a location the FORTRAN language has no way of referencing. The new front end routines provide the user with an offset from a specified array element, and puts this return value into a specific pointer. All information passed to the allocation routine is stored, and this information is then available for other memory handling routines. This feature makes garbage collection, corruption checking and debugging possible.

The information is stored in the data structure laid out below.



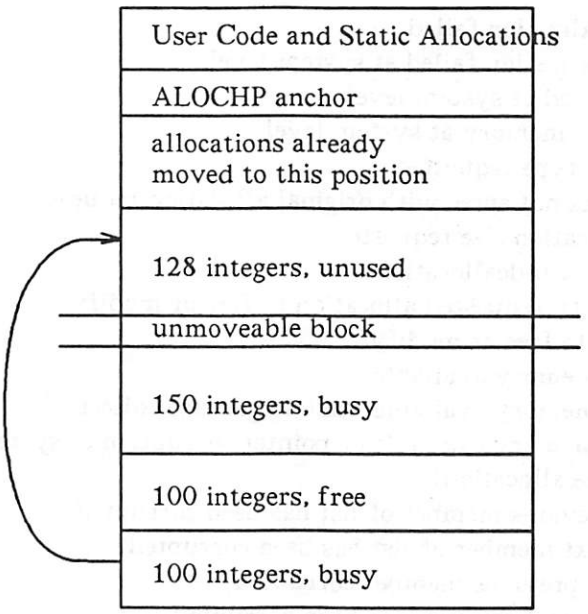
B/F	Pointer to next memory block (used by malloc)
	Pointer to previous member of list
	Beginning address of this block (in bytes)
	Size of user request area (in bytes)
	Location of user base array
	Address of user reference pointer
	Beginning address of user request (in bytes)
	User array declaration type
	Address of first word after user request area
	USER REQUEST AREA
	Back pointer to word preceding user request area
	Pointer to next member of list

4.4 Garbage Collection

The purpose of a garbage collection routine is to turn a fragmented memory layout into a unified structure. The garbage collection routine included in this package works as follows. Initially a list of the allocated (busy) segments is collected and sorted by address. Then, for each allocation in the list, the following steps are performed. First, the distance between the beginning of the allocation and the end of the previous allocation is measured in "user storage units" (e.g. a double precision array should not be moved by an odd number of words). This distance is the size of the unused memory lying between these two allocations, the distance the higher segment can move. The first allocation on the list is the "anchor" of the list; it is set up and used internally, and never needs to be moved, as it was the very first allocation made. Second, the values of the pointers maintaining addresses for this segment are updated. Additionally, when a block is allocated through a call to the allocation routine, the user specifies the index variable that will be used to access this block. The routine stores the address of this variable in the user code, so that garbage collection is now able to update it to point to the first element of the array in its new position. Third, when all the pointers have been moved and updated, the data is transferred.

The first problem encountered in designing the garbage collection routine is that the user may make calls to **malloc** independently from calls to the allocation routine in the package. There are also library routines that call **malloc** without the knowledge of the user. The FORTRAN I/O libraries on the Cray, for example, dynamically allocate buffer areas for opened files at execution time, not statically at compile time. The blocks allocated in this manner cannot be moved; since the location of the pointer to the first element is not saved in these cases, moving the block would be tantamount to losing it. It would also be disastrous for another block to be moved in on top of one of these "unmoveable" blocks. The garbage collector handles this problem by finding all busy, allocated blocks that do not have a corresponding entry in the package allocation list, marking them "unmoveable", and coalescing memory areas around them. When all the memory segments below the unmoveable segment have been collected, the remaining allocations are searched for one that will fit in the space between the last allocation and the unmoveable block.

In the example below, part way through the garbage collection process, the block of 100 integers would be moved below the unmoveable block, as indicated.



4.5 Corruption Checking and Debugging

Since handling dynamic memory allocations is a process of illegal addressing, it becomes more difficult to debug code, as any symbolic debugger or out-of-bounds checker will flag all references to any dynamic area. Debugging routines are useful in helping the programmer detect corruption and errors in the allocated memory areas of a job. Four routines are provided by the current package, each returning detailed error information.

The first checking routine returns the number of members that it finds in the circular list of allocations. If successful, **error** returns with a value of **0**; any other error indicates internal failure, or corruption of the data structure itself, in which case one of the other three routines might be used.

The second routine lays out in table form the addresses, sizes and pointers stored for each of the allocations, essentially those values shown in the last figure in Section 4.3. It returns **0** on success, and an error code on failure.

Another routine examines an individual element of the list, and prints its pointers. It checks for corruption of associated parameters, returning **0** on success and error codes on failure. The fourth routine is similar to the third; however it returns only the error or success value and does not print any additional information. It is, therefore, more useful for execution time error checking than for debugging.

The following errors are tested for and identified in various routines in the package. It was found that sophisticated error identification was one of the most commonly requested parts of the package.

Success
Internal initialization failed
Internal Initialization failed at system level
Allocation failed at system level
Unable to free memory at system level
Invalid array type requested
User type does not agree with original allocation request
Negative allocation size requested
Requesting n < 0 deallocations
Cannot identify requested allocation to free or modify
No members to free or modify
Not enough memory available
Not enough memory available, time to garbage collect
Not enough memory available or pointer corruption at system level
(cannot free allocation)
Pointer to previous member of list has been corrupted
Pointer to next member of list has been corrupted
Pointer from previous member corrupted
Pointer from next member has been corrupted
Area before user area overwritten, check for subscript less than *offset*
Pointer corruption of this member, most likely area before user area overwritten.
Area after user area overwritten, check for subscript greater than *size*

As is see from the above list, as much help as possible is given to the user. The elusive nature of memory problems and the lack of sophisticated interactive debuggers on the Cray make memory debugging extremely difficult. Thus as intelligent error checking as possible should be provided to reduce the need for tedious and time consuming hand tracking of errors.

5. Conclusion

In summary, it is often possible to drastically reduce the storage requirements of a running job. If it is storage that is preventing a job from moving to a physical memory machine, this can be a necessary prerequisite to any further optimization work. The first step is to attack the code of the program with knowledge of the data flow and the perseverance to eliminate surplus storage use wherever possible. Beyond that, the routines and special hardware described in this paper can be of significant additional help.

BUMP
The BRL/USNA Migration Project

Michael John Muuss, BRL
Terry Slattery, USNA
Donald F. Merritt, BRL

The US Army Ballistic Research Laboratory
The US Naval Academy

ABSTRACT

On UNIX systems with many users, or systems that run very large problems, disk space management can be particularly difficult. Space management has generally been accomplished by scripts and programs for determining "disk hogs". Users have been expected to explicitly move their working files to some offline storage media, often using manual procedures and record keeping. In addition, UNIX programs attempting to write to a full file systems get an ENOSPC write error: "No space on device". Frequently, this behavior is not acceptable, especially where programs may execute for a long time.

This paper reports on the implementation of a solution to both these aspects of the disk space management problem, by providing a transparent file migration facility. The result of this software is UNIX filesystems that have the appearance of significantly more capacity than then underlying disk drives, freeing the user community from worrying about managing offline storage media.

The system administrator or a **cron** script may run a utility to cause certain files to be migrated to one of several levels of offline storage. The inode for each migrated file remains present in the filesystem, with special "file handle" data used to recover the file on subsequent access. When a migrated file is opened, the kernel will block the user process, wait for a special user-mode migration daemon to recover the file from backing storage, and then allow the user process to continue. This mechanism is entirely transparent to the user, except for the delay. When a process attempts to write onto a full file system, the system can be configured to block the process, start a "space management" migration function to create file system space, then resume the blocked processes.

The details of the kernel modifications, support daemons, and related software necessary to provide fully transparent file migration will be presented. In addition, the software described is Public Domain, Distribution Unlimited. Several vendors have already expressed plans to incorporate it into their products.

1. Background

Computer systems running UNIX range from personal computers to supercomputers. On systems with many users, or systems that run very large problems, disk space management can be particularly difficult. Historicly, UNIX space management has been accomplished by scripts and programs for determining "disk hogs" and mechanisms for users to explicitly move their files to some offline storage media (magnetic tape, removable disks, etc.), often using manual procedures and manual record keeping.

These traditional techniques have suffered from a number of serious pitfalls, some of which are not at all obvious. The most commonly heard complaint is that users are unclear about how to recover files that have been moved offline, what tape that files are on, how long the tape copies of relocated files will be kept, etc. Vigorous system administrators are often tempted to sweep their filesystems for files that have not been accessed recently, and then move these files to tape. Users are typically notified of this preemptive relocation of their files with a brief note, and typically no serious harm is done. However, if users have been working on a large software project for a long time, and they are using **make** to generate their binary files, the filesystem sweep may determine that many source files are not being used, because they have not even been accessed for a long time. If those files are preemptively moved offline, **make** will explode with error messages the next time it is run, because **make** uses **stat**(2) to build the dependency tree. This is but one example of files that have not been "accessed" in a long time that are still being actively "used".

UNIX programs attempting to write to a full file systems get an ENOSPC write error: "No space on device". Frequently, this behavior is not acceptable, especially in the supercomputer realm where programs may execute for a long time, after which there may not be enough space to write all the results to disk. Generally, it seems that most users would prefer that the system simply waited until additional space became available, perhaps to the accompanyment of warning messages, rather than aborting work in progress. If the out of space condition is recognized by the users, users can often open another window, or use another terminal, or influence a conveniently located colleague, to free up some additional space. It is most unfortunate that work in the process of being written to disk has always been lost in these circumstances.

This paper describes the implementation of a solution to both aspects of the disk space management problem. The software provides transparent file migration and archiving without requiring major changes to the UNIX kernel or filesystem. One result is UNIX filesystems that can have the appearance of significantly more capacity than then underlying disk drives, freeing the user community from worrying about managing offline storage media.

2. Goals

There were two primary goals of this project: First, to create a file migration system for UNIX that provides filesystems that give the appearance of having significantly more online storage than the actual device that contains the filesystem. Second, all unmodified UNIX programs that do not examine the raw filesystem must not be able to detect any difference between regular files, and

migrated files, except for possible delay in completing an open() operation on a migrated file.

Secondary goals were to achieve these features:

1) Have separation between migration *policy*, and migration *mechanism*, so sites that may need to modify the policy will not have to alter the migration mechanism.

2) Make minimal modifications to the UNIX kernel. The basic kernel routines should be installed like a device driver, and all interface "hooks" should be short, and conditionally compiled via #ifdef MIGRATION. Most of the essential functionality should be located in user-mode code.

3) No changes to the size or structure of the on-disk inodes. This permits this software to be installed on machines with existing filesystems, without needing to dump and reload all user files. This also minimizes changes to existing kernel code, dump/restore code, and standalone utilities.

4) Provide support for an arbitrary number and variety of secondary storage devices and recording methods. By using a highly structured and modular interface to the software that handles the secondary storage devices, recipients of this software will be able to easily support additional hardware, and adapt to novel devices, without having to change the fundamental mechanism of this system.

5) Provide extremely robust operation in the face of both system crashes and heavy system use. In essence, this software says "trust me with your files;" that trust must not be violated. Filesystem reliability and availability should be comparable to that of a UNIX system that is not running this software.

6) To have the capability of having multiple copies of migrated files located on secondary storage, for reliability,

7) To have the capability for leaving a copy of a migrated file online, so that either rapid in-migration or rapid space reclamation can be accomplished, depending on which resource is required first, access or storage.

8) To provide support for several types of secondary storage, and for staging files from one form of secondary storage to another. For example, small files might initially be migrated to some type of robotic mass storage, while larger files might go directly to operator mounted magnetic tape. Files on the robotic mass storage might be staged out to magnetic tape if they are not accessed within a few weeks.

Some consciously chosen limitations to this system are:

1) Only regular files can be migrated. It is not possible to migrate directories or special files.

2) To provide migration service only to the machine hosting the disk system. This software is not attempting to provide a CTSS-style "common filesystem" across multiple machines.

3) There is no relief for the problem of creating files that are larger than the online capacity of a single filesystem.

3. Overview

BUMP is a collection of user level tools, supported by a small set of kernel modifications, to provide the user and system administrator with facilities that allow files to be migrated to backing storage and then transparently restored when accessed. These tools allow the user or system administrator to identify files to migrate, force these files into a "pre-migrated" state, copy the pre-migrated files to backing storage, and release the disk storage associated with these files. A specially modified version of the standard ls(1) program will defeat the transparency feature of the migration system, to allow the user to identify files that have been migrated. Additional tools allow the user to recover files in the background for future use, and determine the amount of space taken by files in the migration system. The system administrator has tools available to coalesce sparsely populated migration volumes and to move migrated files between different levels in the storage hierarchy.

To migrate a set of files, the names of the selected files are collected in a migration-list file (see Figure 1, "Functional Diagram"), with an optional hint about future usage and optional comment regarding the reason for migration (e.g. sysadmin forced, etc). All files on this list are migrated to a special "migrate" directory that exists for each filesystem upon which BUMP has been configured to run. Files that have been migrated to the on-disk directory are in a "pre-migrated" state in which the online disk storage has not been released, but the original inode has been changed to mode IFMIG and an entry in the file database has been created. The archiver utility is run to make at least one copy (and typically two copies) of each pre-migrated file onto backing storage media. After the copy has been made, the file is marked as "dual-migrated", because both the online copy and the secondary storage copies exist. From the dual-migrated state, the file can be instantly returned to full online status if the migrated inode is opened, or the on-disk pre-migrated copy may be unlinked to free disk storage if space runs low.

When a migrated file is accessed, either by an attempt to open(2) it, or by an attempt to exec(2) it, the kernel recognizes the migrated inode type, blocks the process attempting the action, and sends a message to a user level daemon requesting that the affected file be reloaded. The daemon runs the archiver utility to copy the file back into the "migrate" directory on the filesystem where the reload is to occur, and once restored to the dual-migrated state, "unmigrates" it. The kernel then learns of the reload completion from the daemon and allows the waiting process to continue.

Free disk space management is automatically provided by a system-wide "low space" disk usage threshold. When the threshold is crossed the kernel notifies the daemon to begin reclaiming free space. The kernel also informs the daemon when file space is completely exhausted, blocking processes that attempt to write on that file system until the daemon creates free space and sends a reply back to the kernel.

The system's operation is easily tailored for specific sites, both in terms of the selection policy for the files to migrate and the type of hardware used for archiving. A hierarchy of storage levels is supported for sites with more than one type of archival media. Policy decisions about which and how many files to migrate are

easily adjustable by the system administrators, making it easy to adapt to varying requirements at different sites.

4. File Finging Tools

The first step in the migration process is to identify candidate files to migrate, perhaps using a policy of selecting files with the largest size or size*age product being selected first. Because selecting files to migrate is independent of the actual archiving mechanism, each site may implement its own selection policy. A combination of the **migsweep** and UNIX **find**(1) utilities can aid system administrators in selecting files based on either the size*age product or some other policy. The only restriction on which files may be migrated is that they be regular files (i.e. type IFREG).

Additionally, the **migsweep** tool will permit each user to designate (in a ".precious" file) the names of files that should never be migrated, up to a certain amount of "permanent" disk storage allocated for that user. In this way, a user can be certain that the ".profile" file and other small files that are very frequently used will not be migrated. Otherwise, getting logged in could become very time consuming!

Users will be provided with another tool to voluntarily migrate files that they know will not be needed for some period of time. The migrate list produced by these tools may include optional fields for a comment and hints about possible future reload times to be used by the archiving mechanism in a multi-level storage hierarchy.

5. Out Migration

The migration file list, kept in a disk file to survive system crashes and reboots, is read by the **migout** tool, that in turn migrates each file to a special per-filesystem 'migration' directory. See Figure 2, "Out Migration (migout)". In this process, migout creates a database entry for each migrated file, allocates the pre-migrated inode in the migration directory, and calls the mig_makemigrated() system call. mig_makemigrated() moves the block pointers from the original file's inode to the pre-migrated inode in the migration directory, zeros the original inode block pointers, stores the file's "handle" in the now empty block pointer area, and changes the original file's type to IFMIG. Files in this state are termed "pre-migrated" because they have been migrated from the normal UNIX filesystem into the special on-disk migration directory, but have not been copied to any other storage media. This operation does not free any storage on the affected filesystem, and in fact, uses another inode for each pre-migrated file. However, it is an atomic operation that results in the file's data blocks being allocated to an inode in a protected directory where advisory file locking may be used to guarantee the integrity of files during archiving.

The "file handle" is a unique sequence number given to each file in the system that allows easy location of all copies of an migrated file, regardless of the storage methods used. File handles are used as the index into the file database to find all occurrences of a migrated file in the archiving system. This indirection is needed to allow files to be moved from one volume to another or from one storage level to another, without having to hunt down and modify the migrated inode to reflect a

change. A file handle is composed of a 32-bit source host identifier and a 32-bit file identifier. Including the source host identifier in the migrated file handle prevents problems from arising when filesystems are accidentally used on the wrong computer system. This could easily happen when removable-media filesystems are present in a site, or it could also happen if a set of dump tapes were reloaded onto a machine other than the originating machine. If the file handle consisted of only a sequence number, and a filesystem from another system was mounted, this would have two undesirable consequences. First, it would grant the requesting user access to protected files owned by other users, and second, having reloaded the migrated files, it would remove the copy of the file from secondary storage, preventing the legitimate owner of the file from reloading it later.

6. Archiving (Staging)

Once the file has been pre-migrated, it must be copied onto at least one backing storage method in order to free the on-line disk blocks. See Figure 3, "Migration Archiving (migarch)." The archiving (or staging) process is provided by **migarch**, a utility that copies files from one storage method to another, including both to and from the on-disk migration directory where pre-migrated files reside. See Figure 4, "Migration Archiving, FROM disk", and Figure 5, "Migration Archiving, TO disk". Migarch will typically be instructed to create at least two copies of each migrated file to facilitate recovery of data written to media that may be subsequently damaged.

A storage method is defined as a type of media (e.g. tape) and a recording format (e.g. ANSI labels). The input to migarch is a list containing the filehandle, destination method, and number of copies for each file to be copied. The file database is searched to find all possible sources of this file, the "closest" copy is determined, an operator request issued for the source and destination volumes to be mounted, the data copied, and the file database updated. This process is repeated for each copy of each file. Destination tape volumes always start out being empty. Migarch is used to coalesce partial volumes, copying files from several partial volumes onto one empty volume. By always writing onto a previously empty tape volume, the system helps guarantee that an existing volume will never be corrupted by unintentional overwriting (such as by power loss during a write operation).

Migarch also allows groups of files to be copied to a single set of volumes, with the selection based on a combination of database parameters such as owner, group, or size. In this case, the input list is built by searching the file database (or the filesystem) for all files matching the desired criteria. One possible use of this feature would be to cause each volume to contain files owned by a single user, for security reasons.

Pre-migrated files that have been archived to backing storage may either remain in the file system or may be removed to free the associated disk space. If they remain in the filesystem, they are called "dual-migrated" files. In this state, one of two operations may occur: 1) a reload request arrives, causing the migin tool to quickly convert the file back to its normal state (see below), or 2) the filesystem runs short of space and the online copy of the dual-migrated file is unlinked to reclaim the disk space it consumed.

7. Modifications to the UNIX Filesystem

In order to implement migrated files, it was necessary to have some way to distinguish between regular files, and migrated files. The most obvious way to achieve this would have been to use another bit in the inode that indicates that the inode is migrated. While the Berkeley 4.2 BSD and 4.3 BSD filesystems have additional space in the on-disk inodes that such a bit could be placed, the current System V filesystems do not have any additional space. However, in both kinds of UNIX systems there are several unused combinations of the IFMT file type bits in the i_mode inode field. Therefore, a single one of these unused combinations is given the symbolic name IFMIG, and is used to mark inodes that are migrated regular files. It is this lack of an additional bit that prevents migrating directory inodes and "special file" inodes, as well as regular files.

When an inode is of type IFMIG, representing a migrated file, it is necessary to store the eight byte "file handle" in the on-disk inode. In the Berkeley 4.2 BSD and 4.3 BSD filesystems there are 16 bytes marked "reserved, currently unused" in the ic_spare[] field, so on the Berkeley implementation the first eight bytes out of the 16 spare bytes are used to store the file handle. The System V on-disk inode has no unused space, so eight bytes of the disk block number array are reused (overloaded) to store the file handle information when the inode is of type IFMIG. This location is referred to as i_fhandle.

There are a number of cases where additional features could have been provided if space existed to store the file handle in all inodes, such as the Berkeley inode format allows. For example, this would have allowed files to have been reloaded for reading only, and then subsequently deleted from the disk, without having to write a new copy to secondary storage. This would also have allowed the concept of a "dual-migrated" file to have been handled in a somewhat simpler manner. A future effort will be to modify the System V filesystem to have the larger inode space required, and then to provide these additional features.

As a result of these file system modifications, it is necessary to update all system utilities that read the raw filesystem. Most notable among these are **fsck**, **dump**, and **restore**.

8. The Definition of Kernel Operations and the Message Protocol

This section describes the communications protocol used between the UNIX kernel mig.c module, and the user-mode migration daemon. For illustrative purposes, interfaces to existing kernel code are drawn from the work done interfacing BUMP to Cray UNICOS 3.0.10 on an XMP. Only the additions are shown, to prevent disclosing any proprietary software. Similar additions exist for 4.2 BSD and 4.3 BSD kernels.

There are two conceptual "layers" to the protocol that is used between the kernel and the migration daemon. The lower level is the basic message-passing mechanism by which the kernel and the user mode daemon exchange chunks of data (messages). The upper level is the definition of the structure of the messages, the meaning of the various message types, and the nature of any expected actions or responses.

8.1. The Message Passing Mechanism

To establish the communication path, the user mode daemon must **open**(2) the special kernel interface device, /dev/mig0. /dev/mig0 will ordinarily be owned by root, and mode 0600, to protect the interface from unauthorized use.

When the daemon opens the /dev/mig0 interface, it receives a normal UNIX file descriptor as the return value from the **open**() sys-call. This file descriptor is used for all subsequent communications. The /dev/mig0 "driver" code has special checks to ensure exclusive use of the interface, ie, only one user mode process may have the Kernel/Daemon message interface open at any one time.

Once the daemon has the interface open, communication between the user mode daemon and the kernel is via normal UNIX **read**() and **write**() system calls.

The daemon's UNIX **write**() system call is vectored through the cdevsw table to the kernel routine migwrite(). The daemon may send a message to the kernel at any time. The driver code has been arranged so the kernel always has one local message buffer to read a message into. Therefore, the kernel will always be able to accept and process a message from the daemon. The byte count argument to the **write**() system call must be exactly the size of one message buffer, ie, sizeof(struct mig_msg), or an EIO error will be returned. A direct consequence of this is that the daemon must perform one **write**() system call for each message sent to the kernel. There is no message "batching" mechanism.

The daemon's UNIX **read**() system call is vectored through the cdevsw table to the kernel routine migread(). The byte count argument to the **read**() system call must be at least as large as the size of one message buffer, so that one entire message can be sent to the daemon in a single operation. Partial reads are not permitted, to simplify the kernel code, and to prevent the daemon from losing track of the start-of-message-buffer location in the byte stream comming from the driver. If the kernel has one or more messages waiting for the daemon, the **read**() system call returns exactly one message back to the daemon, without delay. If the kernel has no messages waiting for the daemon, then the daemon is blocked at interruptible priority until a message arrives.

If the daemon wishes to "sense" the presense of a message, with an optional wait-for-message delay, the UNIX select() call may be used, that vectors through the cdevsw table to the kernel routine migselect(). The migselect() routine indicates that there is read capacity, ie, a message is ready to be read, when that is the case. Attempts to sense the write capacity of the device always return a "true" indication.

When a close() sys-call is performed on the file descriptor returned from the open for /dev/mig0, the kernel routine migclose() will be called. At the time the message passing interface is closed, special cleanup action is taken by the kernel to deal with any messages that the daemon had left outstanding.

Should the the daemon die unexpectedly, or perform an exit() sys-call before closing the file descriptor to the /dev/mig0 interface, the normal UNIX kernel code that cleanly closes all open file descriptors before destroying the hulk of the dead process will ensure that a suitable call of migclose() will occur, even though the daemon process did not explicitly perform one. This will ensure that the exclusive

use semaphore (kernel variable mig_daemon_is_open) is properly cleared, so that when the daemon is restarted, continued operation will be possible.

The contents, organization, and semantics of the message contents are the domain of the higher level.

8.2. The Definition of a Message

The format of the data exchanged between the kernel and the daemon is defined by the C structure "mig_msg", defined in kernel header file migration.h. At present, it looks like this:

```
struct mig_msg {
        int             ms_magic;       /* MIG_MSG_MAGIC */
        int             ms_id;          /* ID of msg, for kernel */
        int             ms_op;          /* operation, see below */
        int             ms_result;      /* may contain errno */
        dev_t           ms_dev;                 /* relevant device */
        struct fhandle      ms_handle;      /* file handle */
        struct mig_inode_id ms_inode;       /* for mig_iunmigrate() */
} mc_msg;
```

The field ms_magic must always be set to the value MIG_MSG_MAGIC, or the message is discarded as "noise", and an error is logged. The field ms_id is a unique message ID that is issued by the kernel. The kernel will never have more than one message outstanding to the daemon with the same message ID. The daemon is required to echo this ID number back to the kernel in any reply message that might be sent. The field ms_op defines the operation, or purpose, of a message. The remaining fields, ms_result, ms_dev, ms_handle, and ms_inode contain valid values only when so noted in the documentation for a specific value of ms_op. Note that the contents of the ms_inode structure are to be considered "opaque" by the daemon, and are intended to be passed intact as one of the parameters to the mig_iunmigrate() sys-call. The daemon should never store or perform any operations on the the ms_inode element.

There are two forms of messages that the kernel can send: blocking messages and asynchronous messages. Blocking messages require a response from the daemon, and asynchronous messages do not require a response. It is important to note that the term "blocking" does not imply that the daemon must answer a message immediately, nor does it imply that the daemon may not answer other messages first. The term "blocking" signifies that there is a user mode process that has been blocked, awaiting the response message from the daemon, and that a kernel message structure remains committed to this transaction until the daemon replies.

There are only two types of messages that the daemon may send, and both are issued in response to a "blocking" kernel message: MIG_D2K_DONE messages, and MIG_D2K_FAIL messages. This simplicity of response messages was intended to simplify the kernel's job when "inventing" proper responses to messages that were outstanding when the daemon closes the /dev/mig0 interface.

Conceptually, either the kernel or the user mode daemon may transmit a message to the other at any time. Messages from the daemon to the kernel will always be processed immediately. Because the generation of kernel to daemon

messages happens in the context of some process other than the daemon process, the message is stored in a message structure, and queued for processing by the daemon. This queueing mechanism requires that there be a pool of message structures, and at present, this pool is of fixed size (kernel parameter MIG_NMSG). When the entire pool of message structures has been committed to use, any additional requests will block waiting for a structure to become available. Therefore, it is desirable to set MIG_NMSG large enough to handle the anticipated number of transactions during "high demand" periods.

8.3. User-settable Modes

There are three new "mode bits" on every process that can be read and set by the user, using the mig_getflag() and mig_setflag() sys-calls.

The first bit, called SPACERETRY, determines kernel behavior when a user executes a write() sys-call to a file on a full filesystem. When SPACERETRY=0, the traditional UNIX behavior occurs, and the write() sys-call returns -1, with variable "errno" set to ENOSPC. When SPACERETRY=1, the write() sys-call vectors to mig_nospace(), that will block the user's process until additional space becomes available, and then allows the write() operation to transparently proceed.

The remaining two bits, called MIGNOTRANSP and MIGCANCEL, control the way the kernel handles an attempt to open a migrated file. When MIGNOTRANSP=0, the user wants transparent access to a migrated file. The process will block until the file has been reloaded, after which the open() will complete normally. If the user process receives a signal, perhaps SIGINT (eg, "^C") when the user gets impatient, the in-migration operation will be permitted to proceed, if MIGCANCEL=0. If MIGCANCEL=1, the daemon will be notified to abort the in-migration operation.

When MIGNOTRANSP=1, the user does not want transparent file access, instead preferring to have the open() return -1 immediately with variable "errno" set to EMIGRATED. If MIGCANCEL=0, the daemon is notified to initiate an asynchronous reload, in the background. If MIGCANCEL=1, no communication with the daemon occurs.

8.4. Reloading Migrated Files

When a user process attempts to open a migrated file, the open() system call is vectored to the kernel routine open(), that calls copen(). copen() contains a block of code to open an existing file, which is supplemented with the following block of code in the Cray os/sys2.c module:

```
#ifdef MIGRATION
        if( !u->u_error &&
            (ip->i_mode & IFMT) == IFMIG &&
            !(mode&FTRUNC) ) {
                extern struct inode *mig_reload();

                /* Note replacement of "ip" after reload */
                if( (ip = mig_reload( ip )) == (struct inode *)0 )
                        return;
        }
#endif
```

where copen() calls the mig_reload() subroutine. On the Cray, similar code is added to the gethead() routine in os/exec.c.

The mig_reload() routine checks the status of the MIGTRANSP and MIGCANCEL mode bits on this process to determine the precise handling of the operation. If MIGNOTRANSP=0 (transparent operation), the daemon is sent a message with ms_op=MIG_K2D_RELOAD_BLOCK, with ms_handle and ms_inode having the file handle and device/inode-number information pertaining to the migrated inode. This message is sent to the daemon using the mig_daemon_wait() routine, which will block waiting for a reply message from the daemon. If the daemon reply message is MIG_D2K_DONE, then the open() proceeds normally. If the daemon reply message is MIG_D2K_FAIL, then the open() fails, returning -1, and the value in the ms_result field of the daemon reply message is used as the value of "errno". The daemon should do all database operations for this message based strictly on the value of the ms_handle element.

If a signal is received while the process is blocked in mig_daemon_wait(), a notification message is sent to the daemon, the signal is posted to the user process, and the open() will return -1, with the value of "errno" being set to EMIGFAIL. The message sent to the daemon will have the same values of ms_handle and ms_inode as the original MIG_K2D_RELOAD_BLOCK message had. If MIGCANCEL=0, then the message sent to the daemon is MIG_K2D_SILENCE_RELOAD, which informs the daemon to proceed with an outstanding RELOAD_BLOCK operation, but not to send any further notification back to the kernel, ie, treat the operation as if it had originally been of type MIG_K2D_RELOAD_ASYNC. If MIGCANCEL=1, then the message sent to the daemon is MIG_K2D_CANCEL_RELOAD, which informs the daemon that the reload of this file is no longer required, and if possible, it should be aborted. If the daemon has already "committed" to the reload operation, no harm is done by allowing the reload to complete.

In this protocol, there is the a potential for a non-harmful race condition between the kernel queueing a SILENCE or CANCEL message to the daemon just before the daemon begins sending a DONE or FAIL message to the kernel for that operation. Therefore, for best results the daemon should always check to see if there are any additional kernel-to-daemon messages waiting, before the daemon sends reply messages to the kernel. When the race condition exists, the daemon will send a reply message to the kernel that is no longer expected. This will be detected in the kernel routine mig_process_response(), and in this case, the kernel will simply note the occurrence of the race by sending the daemon a MIG_K2D_UNEXPECTED message for adding to the daemon log files. In this message, the original message is duplicated for return, with ms_id being the new message ID ms_result being set to the ms_id field of the unexpected message just received, and ms_op being set to MIG_K2D_UNEXPECTED. If the kernel is unable to obtain a message buffer to log this error condition, then a console printf() is performed, with the message "WARNING: mig_process_response: unexpected daemon reply, id=%d". This behavior was necessary on the out of buffers condition to prevent a potential buffer deadlock condition.

If MIGNOTRANSP=1 (non-transparent operation), the open() fails with "errno" set to EMIGRATED. If MIGCANCEL=0, then the daemon is sent a MIG_K2D_RELOAD_ASYNC message, with ms_handle and ms_inode having

the pertinent information. This permits the daemon to initiate an asynchronous reload operation for the file. The daemon should do all database operations for this message based strictly on the value of the ms_handle element. It is anticipated that the daemon would probably queue these asynchronous requests at a lower priority level than requests where there is a user process actively waiting a reload operation. If MIGCANCEL=1, then no daemon notification happens at all.

If the daemon is not running when mig_reload() is called, the open() fails, and "errno" is set to EMIGOFF. No access to a migrated inode is permitted while the daemon is not running.

8.5. Low Space and No Space

There is a system-wide disk usage threshold mig_minfreefrags that can be read by anyone using the mig_getflag() sys-call, and can be set by the superuser with the mig_setflag() sys-call.

It is planned that in a future version of this software, where tighter integration with existing system utilities would be possible, that the minimum space threshold would be settable on a per-filesystem basis. On Berkeley UNIX systems, this would most likely become part of the in-core mount table information, set by an extra field in /etc/fstab.

When a filesystem transitions from an amount of free storage above the threshold to an amount below the threshold, the routine mig_lowspace is called. Here is the code fragment from the Cray fs/c1/c1alloc.c routine:

```
#ifdef MIGRATION
{
    extern int mig_minfreefrags;
    if( fp->s_tfree >= mig_minfreefrags &&
        (fp->s_tfree - reqblks) < mig_minfreefrags )
          mig_lowspace(dev, fp->s_tfree);
}
#endif
```

The mig_lowspace() routine sends a message to the daemon, with ms_op set to MIG_K2D_LOWSPACE, ms_dev being the device running low on space, and ms_result set to the amount of storage remaining. No daemon response to the kernel is expected.

On receipt of the LOWSPACE message, the daemon has the option (depending on configuration parameters) of initiating a Space Management function to make additional space on the device. This might include removal of some pre-migrated files, and/or the initiation of an immediate filesystem sweep and out-migration operation, depending on the site-specific configuration.

Note that the daemon has to be prepared for getting several LOWSPACE messages concerning the same device within a short period of time, as the storage level oscillates around the threshold level. These should be collapsed into a single event within the daemon.

When an attempt is made to allocate blocks on a filesystem that is full, the routine mig_nospace() is called. Here is the code fragment from Cray fs/cl/clalloc:

```
#ifdef MIGRATION
        if( mig_nospace( dev ) == 0 )
                goto retry;
#endif
        prdev("ERROR: alloc.c: no available free space",dev);
```

mig_nospace() first checks the setting of the process SPACERETRY mode bit. If SPACERETRY=0, the kernel calls mig_lowspace() with a space level of zero to note the condition, the error ENOSPC is returned from mig_nospace(), and allocation fails.

If SPACERETRY=1, then a message is sent to the daemon with ms_op=MIG_K2D_NOSPACE and ms_dev set to the relevant major/minor device code. The kernel then blocks the user process until a reply message is received from the daemon. If the response is MIG_D2K_DONE, then the storage allocation is retried. There is no race condition here, because if another process has consumed the newly made storage before this process retries the allocation, mig_nospace() will be called again, and the operation repeats. If the response from the daemon is MIG_D2K_FAIL, then the allocation operation is abandoned, and an ENOSPC error is returned to the user process. It is not anticipated that the daemon would ever return a MIG_D2K_FAIL code to a NOSPACE message, as that defeats the purpose of this feature.

If the user process fields a signal while it is blocked waiting for the daemon reply to the NOSPACE message, then the kernel sends the daemon a message with ms_op=MIG_K2D_CANCEL_NOSPACE and ms_dev set to the major/minor device code. Note that the daemon is still expected to act on the no space condition, even though the user process is no longer blocked waiting on space, ie, treat the message as a NOSPACE message with 0 blocks left. Perhaps a better name would have been SILENCE_NOSPACE. There is a potential race condition here, as with the MIG_K2D_RELOAD_BLOCK message described earlier; it is not harmful, and mig_process_response() will take the same remedial action.

Note that the daemon should be prepared for multiple processes to encounter a NOSPACE condition on a given device within a fairly short time of each other. The daemon is responsible for holding all of them, and not releasing them until it is known that some storage becomes available, which the daemon can learn about from (a) the Space Management task on that device completing, and (b) periodic sampling of free space levels.

If mig_nospace() is called and the user has set SPACERETRY=1, but the migration daemon is not running, rather than returning an error, the kernel adopts a simple retry strategy, to prevent the user process from seeing unwanted ENOSPC errors. If the user has no signals pending, a non-interruptible kernel sleep will be initiated in mig_nospace(), with a one minute timeout. This will cause an allocation retry once a minute, until additional storage becomes available, or the process receives a signal, perhaps SIGINT (eg, "^C") when the user gets impatient. This polling behavior is not optimal, especially if several dozen processes need additional

space, but it prevents the very desirable SPACERETRY feature from evaporating when the daemon is not running.

Thanks to this feature, processes that have SPACERETRY=1 should never see an ENOSPC error return. This is extremely valuable for long-running processes that write all their answers to disk just before exiting.

8.6. Truncating Migrated Files

Whenever a process attempts to truncate a migrated file to zero length, the routine mig_trunc() is called. Truncations to non-zero lengths cause a normal mig_reload() operation, as described above. The code fragment from Cray fs/cl/cliget.c. Also, if it is desired to take "core dumps" on top of migrated files, a similar modification will be required in os/sig.c

```
#ifdef MIGRATION
        /* Reload the file prior to truncation, if new size > 0 */
        if((ip->i_mode & IFMT) == IFMIG)  {
                if( size == 0 )
                        u->u_error = mig_trunc( ip );
                else
                        u->u_error = mig_reload( ip );
                if( u->u_error )
                        return;
        }
#endif
```

When a migrated file is truncated to length zero, the kernel sends a message to the daemon with ms_op=MIG_K2D_TRUNCATE and ms_handle set to the appropriate file handle, after which, the kernel converts the inode back into a regular file (i_mode=IFREG) with length zero, and the normal truncate operation proceeds.

Typically, for migrated files whose data resides on some form of backing store, the daemon would move the database entries for the backing store copies into an "age, then queue for volume reclamation" queue, for subsequent Volume Management operations.

If the daemon is not running when mig_trunc() is called, the truncate operation fails, and "errno" is set to EMIGOFF. The contents of migrated inodes may not be altered while the daemon is not running, to prevent the migrated file database from becoming inconsistent with the state of the filesystem.

8.7. Unlinking Migrated Files

Whenever a process attempts unlink a migrated file, the routine mig_unlink() is called. The code fragment from Cray os/sys4.c, routine unlink():

```
#ifdef MIGRATION
        /*
         *  If this is the very last link to a migrated file,
         *  inform the migration system, and allow it the opportunity
         *  to note (and perhaps refuse) the operation, before
         *  removing the directory entry or dereferencing the inode.
         */
```

```
        if( ip->i_nlink == 1 && (ip->i_mode & IFMT) == IFMIG )  {
                if( (u->u_error = mig_unlink( ip )) != 0 )
                        goto out;
        }
#endif
```

When the last link to a migrated inode is removed, the kernel sends a message to the daemon with ms_op=MIG_K2D_UNLINK and ms_handle set to the appropriate file handle, after which, the kernel converts the inode back into a regular file (i_mode=IFREG) with zero size, and the normal unlink operation proceeds.

Typically, for migrated files whose data resides on some form of backing store, the daemon would move the database entries for the backing store copies into an "age, then queue for volume reclamation" queue, for subsequent Volume Management operations.

If the daemon is not running when mig_unlink() is called, the unlink operation fails, and "errno" is set to EMIGOFF. Migrated inodes may not be removed while the daemon is not running, to prevent the migrated file database from becoming inconsistent with the state of the filesystem.

8.8. The System Call Interface

In addition to the message passing interface, the kernel support for the migration system also provides several additional system calls.

In the present implementation, it was decided that these system calls would not be coded using the normal kernel sysent[] table, but would be handled by a private mechanism, so as to minimize the amount of existing kernel code that would have to be altered, and to prevent having to make vendor-specific system call interface modules for inclusion /lib/libc.a, the C runtime library. When a vendor installs this code in their system, they would presumably assign system call numbers, and add the interfaces to the C runtime library, for slightly increased speed and clarity.

To the applications programmer, the new migration system calls are indistinguishable from direct system calls. In the remainder of this paragraph, the details of the present implementation will be discussed, and then the difference will be ignored henceforth. All use of these new system calls is expected to be via the library routines provided in libmig.a, which establishes contact with the new system call interface in the kernel by opening device /dev/mig1, that will ordinarily be mode 0666 to permit general use. The interface routines in libmig will bundle up the system call number and arguments into a buffer, and write() it to /dev/mig1. Kernel routine migwrite() will copy this buffer into kernel space, and call the routine indicated by the system call number. The interface into the migration system call routines is identical to the interface seen when calling via the sysent[] table, to permit easy conversion to the direct method.

8.9. Flag Manipulation

```
mig_getflag( cmd, pid )
int     cmd;
int     pid;
```

```
mig_setflag( cmd, pid, value )
int     cmd;
int     pid;
int     value;
```

There are presently four kernel parameters that may be read or altered using these two system calls. The SPACERETRY, MIGNOTRANSP, and MIGCANCEL parameters are one-bit quantities, and are stored on a per-process basis. The THRESHOLD low space threshold is presently a single integer value that applies to all filesystems (see LOWSPACE remarks, above). The "pid" argument must specify a live process, to avoid an ESRCH error. A "pid" value of zero is interpreted to indicate the pid of the process performing the system call. If the process specified by "pid" belongs to a different user, and the process performing the system call is not running as the superuser, then an EPERM error is returned.

When cmd=MIG_FLAG_SPACERETRY, the one bit parameter SPACERETRY is read or written. With a value of 0, writing to a full filesystem returns an ENOSPC error, while with a value of 1, writing to a full filesystem will always succeed, although potentially with some delay until free storage is available.

When cmd=MIG_FLAG_NOTRANSP, the one bit parameter MIGNOTRANSP is read or written. When cmd=MIG_FLAG_CANCEL, the one bit parameter MIGCANCEL is read or written. See the earlier remarks on reloading migrated files for a detailed description of the effects of these bits.

When cmd=MIG_FLAG_THRESHOLD, a mig_getflag() will return the current threshold upon which the daemon will be notified of a low space condition. Only the superuser may alter this parameter with mig_setflag(); non-privileged use will result in an EPERM error. Only non-negative values are permitted. Note that in this case the value of "pid" has no significance, but must be valid.

The SPACERETRY, MIGNOTRANSP, and MIGCANCEL bits are carried in the process structure p_flag word. They are inherited across forks, so that all child processes run with the same storage and migration behavior as the parent. Typically, a user would set the desired operating mode of his shell, and then all processes will behave in the desired manner. In order to permit these bits to be inherited by child processes, a one-line change needs to be applied to the fork() routine. In the subroutine newproc() in Cray os/fork.c:

```
#ifdef MIGRATION
        rpp->p_flag |= (rip->p_flag & (SRTIM|SCPUS|
                        SSPACERETRY|SMIGCANCEL|SMIGNOTRANSP));
#else
        rpp->p_flag |= (rip->p_flag & (SRTIM|SCPUS));
#endif
```

8.10. Creating a Migrated Inode

```
mig_makemigrated( source, dest, handle )
char    *source;
char    *dest;
struct fhandle *handle;
```

This system call is intended for use by the out-migration tool. Only processes running as superuser may use this system call; other users will get an EPERM error. In ordinary use, the "source" file will be some user file, and the "dest" file will be in the migration directory for that filesystem.

The file named by "source" must already exist and be a regular file (i_mode=IFREG), and the file named by "dest" must not yet exist. Both must be on the same filesystem. The destination file is created in mode 0400, and is given the same size as the source file. Then, all of the disk blocks are transferred from the source file to the destination file, and removed from the source file. Finally, the source file is changed from file type regular (i_mode=IFREG) to migrated (i_mode=IFMIG), and the file handle given in "handle" is stored in an implementation-specific location within the source file's inode. Note that the source file retains it's original ownership, access modes, size indicator (i_size), access and modification times. However, it now no longer is using any storage for disk blocks. Any attempt to access this file will result in notification of the migration daemon, as described in the previous section.

8.11. Unmigrating an Inode

```
mig_unmigrate( source, dest )
char   *source;
char   *dest;

mig_iunmigrate( source, idest )
char   *source;
struct mig_inode_id *dest;
```

Two system calls exist for reversing the effect of the mig_makemigrated() system call. The mig_unmigrate() form takes two path names, and is intended for human-driven diagnostic and disaster-recovery uses, and is not used by the production migration software. The mig_iunmigrate() form uses an opaque object of type "mig_inode_id" such as is found in the ms_inode field of kernel to migration daemon messages like MIG_K2D_RELOAD_BLOCK.

In ordinary use, the "source" file will be in the migration directory, and the "dest" file will be the corresponding migrated user file. The "source" file must be a regular file (i_mode=IFREG), and the "dest" file must be a migrated file (i_mode=IFMIG). Both files must be the same size. The "dest" file is converted back into a regular file, and then the disk blocks are moved from the "source" to the "dest" file. At this point, only the "change" time on the "dest" file will have been affected by the mig_makemigrated() and mig_iunmigrate() process; all other inode fields will be exactly as they were before the inode was migrated. Finally, the "source" file is unlinked.

8.12. File Status

One of the goals of this project was to implement a file migration capability that was so transparent that, except for additional delays for moving files in from backing storage, there would be no user visible differences. One implication of this is that the stat() system call must not indicate that a file is migrated -- otherwise, every application program that looks at the i_mode field of the inode (eg, find(1), du(1), etc) would have to be modified to know about the new inode type, IFMIG.

That would not have been transparent at all! Instead, the stat() system call has been modified so that migrated inodes seem to be of regular file type, IFREG. This required the following change to Cray os/sys3.c routine stat1():

```
#ifdef MIGRATION
        /*
         *  Migrated files look like regular files to all users.
         *  Programs that care about the difference should use
         *  the mig_stat() sys-call instead.
         */
        if( (ip->i_mode & IFMT) == IFMIG )
                ds.st_mode = (ip->i_mode & ~IFMT) | IFREG;
#endif
```

Having made this modification, this raises the question of how a program that desired to know the true status of an inode can obtain it. This lead to the creation of two additional system calls:

```
mig_stat( name, statp, handle )
char            *name;
struct stat     *statp;
struct fhandle          *handle;

mig_lstat( name, statp, handle )
char            *name;
struct stat     *statp;
struct fhandle          *handle;
```

mig_stat() functions exactly like stat(). For non-migrated files, the file handle structure contains all zeros, while for migrated files, the file handle structure is non-zero, and contains the appropriate migration information copied from the inode. Note that in the later case, the file type will still be IFREG, not IFMIG, so that code that may be handed the stat structure would not have to be concerned with the extra file type.

For kernels that support the Berkeley concept of a symbolic link, the mig_lstat() subroutine is to the lstat() system call, as mig_stat() is to the stat() system call.

8.13. Error Codes

The existing set of kernel error codes that system calls can return in "errno" have been supplemented by several error codes that are specific to support for the migration system.

errno=EMIGRATED is returned when a user attempts to access a file that has been migrated, and the user has requested non-transparent access. This error is the result of the system honoring the request for non-transparency, and does not signify any difficulty.

errno=EMIGFAIL is returned when a wait for a transparent migration operation is interrupted by a signal.

errno=EMIGOFF is returned when a user attempts to access or delete a file that has been migrated, and the migration daemon is not running. No accesses to

migrated files are permitted until the migration daemon has been restarted, so that the migration database remains consistent with the filesystem. The user should contact an operator or system administrator to have the daemon restarted.

errno=EMIGNLOC is returned on an attempt to migrate a file between two different filesystems, or when the source file is not local to the executing machine, eg, is an NFS file on a remote server. This error only occurs in the mig_makemigrated() and mig_unmigrate() system calls, and should only be seen by superuser processes.

9. The Migration Daemon

Consistent with the principles of good modular operating system design, and in order to keep the required kernel additions to a minimum, most of the real work to handle file reload operations and out of space conditions is delegated by the kernel to the user-mode migration daemon process. In turn, the migration daemon itself does very little more than prioritize and queue requests from the kernel, and spawn various other processes to execute the needed migration tools. In particular, the daemon can spawn multiple copies of the in-migration tools, and it can also initiate a space management procedure (typically a shell script) for every filesystem that is running low on available disk storage.

When the migration software is installed on a machine, the migration daemon becomes an integral part of the operating system software on that machine. The migration daemon plays the same kind of critical role with the UNIX kernel filesystem functionality as the Internet "super-server" **/etc/inetd** and the domain name server **named** play for the UNIX network functionality. In ordinary operation, the migration daemon is not expected to die. However, if the daemon should die (or be killed), the kernel will make reasonable responses to all filesystem requests that are made. In particular, if a filesystem runs out of space when the daemon is not running, SPACERETRY=1 operation for reliable file writing is still provided, using a simple, less efficient all-kernel technique. If the migration daemon is not running, users will be prohibited from opening or deleting migrated files, although operations that affect only the inode will still be permitted, such as **chmod**(2), **rename**(2) or **mv**(1). This behavior is necessary to keep the migration system databases synchronized with the state of the filesystem. If the migration daemon is not running, the system should be considered to be experiencing a serious problem. Fortunately, it should always be possible for the superuser to log in on the console to take remedial action. This implies that crucial system files such as **/bin/sh** should not ever be migrated. It would be wise if the main system directories **/bin**, **/lib**, /etc, and **/usr/bin** were always exempted from out-migration. It would be better still if the entire root and **/usr** filesystems were never subjected to out-migration. Not only will this keep the system more responsive at a very small penalty in online storage used, but it will also ensure that all files needed for effecting system repairs will be available online when such repairs are called for.

10. In Migration

Reload requests are sent from the kernel to the daemon using the protocol described earlier. The migration daemon will fork and start a **migin** process to cause the file to be reloaded. See Figure 6, "In-Migration Function (migin)". **migin**

runs the **migarch** utility to copy the file back into the file system if it is not already in place (i.e. dual-migrated). When the file is reloaded into the appropriate file system's migration directory, the daemon performs a mig_iunmigrate() system call, and notifies the kernel of the successful reload. If **migarch** experiences unrecoverable errors while trying to read every one of the multiple copies of the migrated file, then errors are logged in the migration log file, and an appropriate error is return through the daemon to the user process.

Files that are online in the filesystem have no existence in the secondary storage, because of the inability to store a file handle in the inode of a regular (i_mode IFREG) file. Therefore, when a file has been reloaded, all of the copies on secondary storage should be considered obsolete. However, to provide disaster recovery, the secondary storage copies of the migrated file can not immediately be turned over to Volume Management for reclamation. Instead, the secondary storage copies must be aged for a minimum of twice the backup interval before the storage can be reclaimed. If a file is removed, and then the filesystem is reloaded, it will still be available for a few additional days. Thus, any file that has been reloaded will be marked as obsolete in the file database, but will continue to be available for disaster recovery until the volume on which it is stored is reclaimed.

11. Space Management

To enable fully automatic recovery from disk space shortages, the migration system utilizes two kernel-to-daemon messages. The LOWSPACE message is sent when the freespace on a filesystem falls below the configured threshold and the NOSPACE message is sent when a filesystem is full. Upon receiving either of these messages, the daemon starts the **migspace** utility, that implements a system-specific policy for creating additional space. See Figure 7, "Space Management (migspace)". Typically, this policy would be to first consult the migration database to see if the affected file system has any dual-migrated files whose online disk storage can be immediately reclaimed. If this does not provide a sufficient amount of space, a list of new migration candidates should be built, and an out-migration operation would be initiated to move these files to backing storage. The worst case can occur when a few large files fill an entire filesystem, requiring all other files to be migrated to secondary storage.

Conservative sites, or sites that do not have 24-hour operator coverage may choose to configure the **migspace** script to create space only by unlinking all dual-migrated files, and then to give up and wait for human intervention. In this case, processes needing additional file space will be blocked until additional space becomes available, or some human kills them.

Note that some care needs to be taken to ensure that **migout** processing gets priority access to the tape drive in a single tape drive system.

12. Databases

There are several databases and intermediate files used by the BUMP tools; 1) a file database to map file handles into file location, 2) a volume database to identify the location and type of a storage volume, 3) a list of files to be staged from one archiving method to another, containing source file handle, number of copies to make, etc. All databases and intermediate files use the same basic format and are

manipulated by a common set of routines. The format is an ASCII text file, with each record newline terminated and each field terminated with a vertical bar "|" character. Database fields that may require updating occupy a preset number of characters, so that the field may be updated in place. Concurrent update is prevented by using the appropriate kernel file locking features.

Using a simple file format for the databases which can be manipulated by the standard UNIX text processing tools results in a significant economy. Everything from simple enquiries up to the most sophisticated queries may be resolved using simple combinations of grep, awk, ed, and the rest of the UNIX text processing tools. Creating management reports on storage utilization can be handled with small Shell scripts that are easily tailored to the specific needs of individual sites. Using a text file format also permits ordinary text editors to be used to examine and modify the databases during development. It is anticipated that this convenience will prove similarly useful when disaster recovery is required.

Performance of this simple strategy is not anticipated to be a factor. If each database record requires 100 bytes, then the information for 10,000 migrated files will use a single megabyte of storage for the database. A large system may expect to have several hundred thousand migrated files at any time, totaling perhaps hundreds of gigabytes of storage, yet the migration database will remain comparatively modest in size.

13. Philosophy & Implications

The illusion of having unlimited on-line disk storage can be a great convenience for users. However, files may be migrated to automatic devices with recall times that can be measured in fractions of minutes, and files that have been migrated to devices that require operator intervention, such as conventional magnetic tape, will typically require several minutes per recall. The trade-off between the convenience of having extra storage and additional delay is certain to have uneven user appeal. Assuming that the file migration system has been well implemented, the success or failure of the file migration system in a particular environment will depend on having a migration policy that suits the needs of the most important users. This is why there has been such a strong emphasis placed on separating the migration *policy* from the migration *mechanism* — because no single migration policy will be able to meet the needs of all sites.

The most challenging environment to implement a successful migration policy in is unfortunately the environment that UNIX usually excels at: the highly interactive timesharing environment. Balancing the need for high interactivity with the need to have significantly more online storage than the actual capacity of the underlying filesystem will be difficult. Success is likely only if (a) the users perceive the benefits of additional convenient file storage as outweighing the inconvenience of an occasional delay in file access, and (b) the file migration policy has been tuned so that an average users "working set" of files is not ordinarily selected for migration, so that reload delays are incurred infrequently.

It is important to distinguish between the functions of the file migration system, the filesystem backup system, and the private user tape handling facility (sometimes also called an "archiving system", a usage that conflicts with the usage in this paper). Some operating systems, such as Cray's COS operating system,

attempt to integrate parts of all three functions into the general filesystem capability. It is the purpose of this project only to provide a file migration capability, and not to disturb or significantly alter existing backup and user tape handling conventions. The file migration system is intended to provide the appearance of UNIX filesystems that are significantly larger than the actual capacity of the disk hardware. The filesystem backup system is intended to provide disaster recovery, so that a failed disk drive can be restored to the same state that it had at the time the last backup tapes were written. The backup system is also often used to help recover from serious user errors, where files are inadvertently deleted. When the deleted files have not been modified since the last backup tapes were written, then the files can be restored from the backup tapes, and the user error is undone. Finally, the user tape handling facility is intended to allow users to selectively but permanently save files that may be needed again in the future, but will not be needed online for a significant amount of time.

To prevent filesystem backups from taking an absurd amount of tape, and an even more absurd amount of time, it is necessary to modify the backup procedures to discriminate between regular files and migrated files. For migrated files, the backup procedure should record only the inode information onto the dump tapes, and not the actual contents of the file. This is consistent with the definition of the backup system as providing only disaster recovery. Note that it is the fact that a filesystem may be reloaded from backup tapes that forces the secondary storage copies of all migrated files to be retained for an aging period. This ensures that if the most recent backup is reloaded onto the disk, the secondary storage copies of recently deleted files will continue to exist for a brief additional time, so that they can be reloaded by the user if needed.

The migration system does not keep a consistent set of file names on the secondary storage media, because all activity is identified by the inode number and file handle, and because one inode may have many different path names that lead to it. While the migration system does record a complete copy of the information in regular file inodes onto the secondary storage, it does not record any path name information, because directories and special files can not be migrated. As a result, the migration system is not suitable for use as a filesystem backup mechanism. If a filesystem was entirely destroyed and no dump tapes were usable, enough information is stored in the migration system so that all migrated files could be recovered, but all files would appear in their owners home directory.

Having the file migration capability does not relieve the need for a user tape handling facility, nor does it permit users with large storage requirements to ignore their responsibility for monitoring their storage usage and managing their collection of private tapes. The file migration system merely changes the point at which the filesystem appears to be full. The limiting factor will shift from the number of disk blocks available on a filesystem to the number of inodes available on a filesystem. Most UNIX filesystems have a fixed number of inodes allocated per filesystem at the time that the filesystem is initially created (with **mkfs**). Thus, the additional inode requirements must be taken into account when filesystem are created. Many more inodes will be needed on filesystems that will store migrated files; 200,000 inodes is a good minimum for larger filesystems (0.5 Gbytes to 1 Gbytes).

Even those few UNIX systems that can dynamically allocate additional inodes when needed are still limited by the fact that the size of the disk provides an upper bound on the number of files on the filesystem. This seemingly contrived limiting case becomes more of a serious issue when online storage for at least one copy of the largest user file needs to be held in reserve from the disk full of inodes. Failure to observe this relationship could prevent users from retrieving their largest files, and in general would probably be preceded by the migration system being forced into severe "thrashing" of files between primary and secondary storage.

Therefore, when implementing the file migration system on a particular machine, it is strongly recommended that the administration adopt a two-part policy on file storage, and that the user community be advised of this policy before the file migration system is activated. The first part of the policy is to note that when online space is required, user files may be transparently moved to secondary storage, following the detailed rules of the migration policy elected by the site. The second part of the policy is to note that files may not be maintained in the migration system, even on secondary storage, for more than a period of 1.5 years (or a similar time limit), and that users who require longer term storage of their data must take advantage of the user tape handling facility that has been provided. An automatic tool will be provided that will, with suitable advance warning to the users via E-mail, enforce this policy.

14. Current Status

As of this writing, the kernel support for the migration system is complete, and has been well tested. A demonstration package which includes **migout**, **migin**, and the migration daemon has been assembled that allows files to be migrated and unmigrated. Implementation of the **migarch** utility and the set of methods for tape-style devices is well underway. Overall, most of the pieces now exist, and the final work of assembling the high-level tools is progressing well.

It is anticipated that this software will be in full production status in BRL in the early Fall of 1988, and will be made generally available as Public Domain Distribution Unlimited software by Fall/Winter of 1988.

15. Future Work

The task of moving a user from one filesystem to another filesystem, to more evenly balance storage requirements is a task that system administrators will occasionally have to perform. Typically, this is accomplished on Berkeley systems using back-to-back **tar** programs, eg,

```
cd fromdir; tar cf - . | (cd todir; tar xf -)
```

and on System V machines, this is done using a combination of **find** and **cpio**, eg,

```
find . -depth -print | cpio -pdlm
```

If a user with a significant number of migrated files was to be relocated to a new filesystem using this technique, the correct effect would be achieved, but the migration system would engage in a significant amount of activity. All the files on the source filesystem would have to be migrated in to the original disk, that would probably cause the space management function to have to out-migrate other files to make room. Then, all those files would be copied to the destination filesystem,

that would probably also cause the space management function to have to out-migrate files there, too. The result of this is that whole collection of old files will have been brought back onto disk on a new filesystem, where they will consume online storage until they have aged sufficiently to qualify for out-migration. Adequate kernel support exists to permit the implementation of a **cpio -p** substitute that would relocate migrated inodes without forcing a reload operation, but as of this writing, this has not yet been done.

For the majority of UNIX systems, the most popular secondary storage medium for file migration is likely to be operator mounted reels of magnetic tape. This makes the interface to the operator an important part of the file migration software, so that it is easy for the operator to determine what tape is to be mounted, and on which drive. In order for the migration system to be effective, this must work well, or the extra delay in mounting tapes will reflect poorly on the migration system. However, the design and implementation of a good operator interface mechanism for UNIX is properly the subject of another project. By necessity, the initial version of the file migration software will offer a very simple interaction with the operator, except on systems like the Cray where an operator interface package already exists. Then, as time progresses, a separate effort to implement a good, portable operator interface package will be initiated, with the goal being to release another piece of software into the public domain. For the present, this has not been done.

Many additional migration methods will be added as the project progresses, with network tapes, Masstore systems robotic cartridge tape units, and the 8mm Exabyte tape cartridges being prime candidates.

The current design accomplishes its goal of no filesystem inode changes; however, this prevents certain worthwhile features from being provided. For example, a desirable feature is the ability to reclaim space used by files that have been reloaded for read-only access, by taking advantage of the copy on secondary storage. Another is to be able to supply the first portions of a file to a reading process while the balance of the file is being reloaded. These features require that the filehandle and quantity of reloaded data be added as new fields in the filesystem inodes. Future work will incorporate these filesystem changes and provide enhanced functionality.

In order to minimize the impact on the kernel, and various ancillary programs such as **mount**, the current implementation uses a single low space threshold, expressed in blocks, for all mounted file systems, regardless of file system size. A much better strategy would be to read the low space threshold from a file such as /etc/fstab, and provide it to the kernel as part of the **mount**(2) system call. This would allow the low space threshold to be set independently for each filesystem.

In a network filesystem environment, such as provided by Sun NFS, the migration software will reside entirely on the file server machine, so that file reloading will be entirely transparent to the client machines. It is presently unknown whether there will be any interactions between the potentially lengthy time required to open a migrated file, and protocol timeouts in the client machines. It may also be necessary to increase the number of **nfsd** daemons to account for some of them being blocked on migrated file opens.

Presently, there is no way to add the functionality of the new system calls mig_stat() and mig_lstat() to the current NFS protocol. One implication of this, combined with the fact that the file server handles the migrated files, is that there is no way for a client machine to determine whether a file is migrated or not. Finally, it is not clear how to propagate signals on the client to the server. If the an open of an NFS file is blocked on the server due to a migration reload operation, the signal needs to travel over the network. All these issues will be investigated in detail after the software is operating well for machines with locally attached disk drives.

16. Conclusions

With a very limited set of modifications to the UNIX kernel, it is possible to provide a fully transparent file migration capability, preserving the complete semantics of the UNIX filesystem. Built around these kernel capabilities are a highly modular set of utility programs that select files for migration, and implement the details of moving files between different devices. One of the best aspects of the design is the strong separation between *policy* and *mechanism*, so that the special needs of individual sites can be satisfied with a single software mechanism.

This software promises to satisfy a need that large scale users of UNIX systems have long felt, yet provides the capability in such an elegant and transparent manner that even the most discriminating UNIX person should not bristle. Much.

Acknowledgements

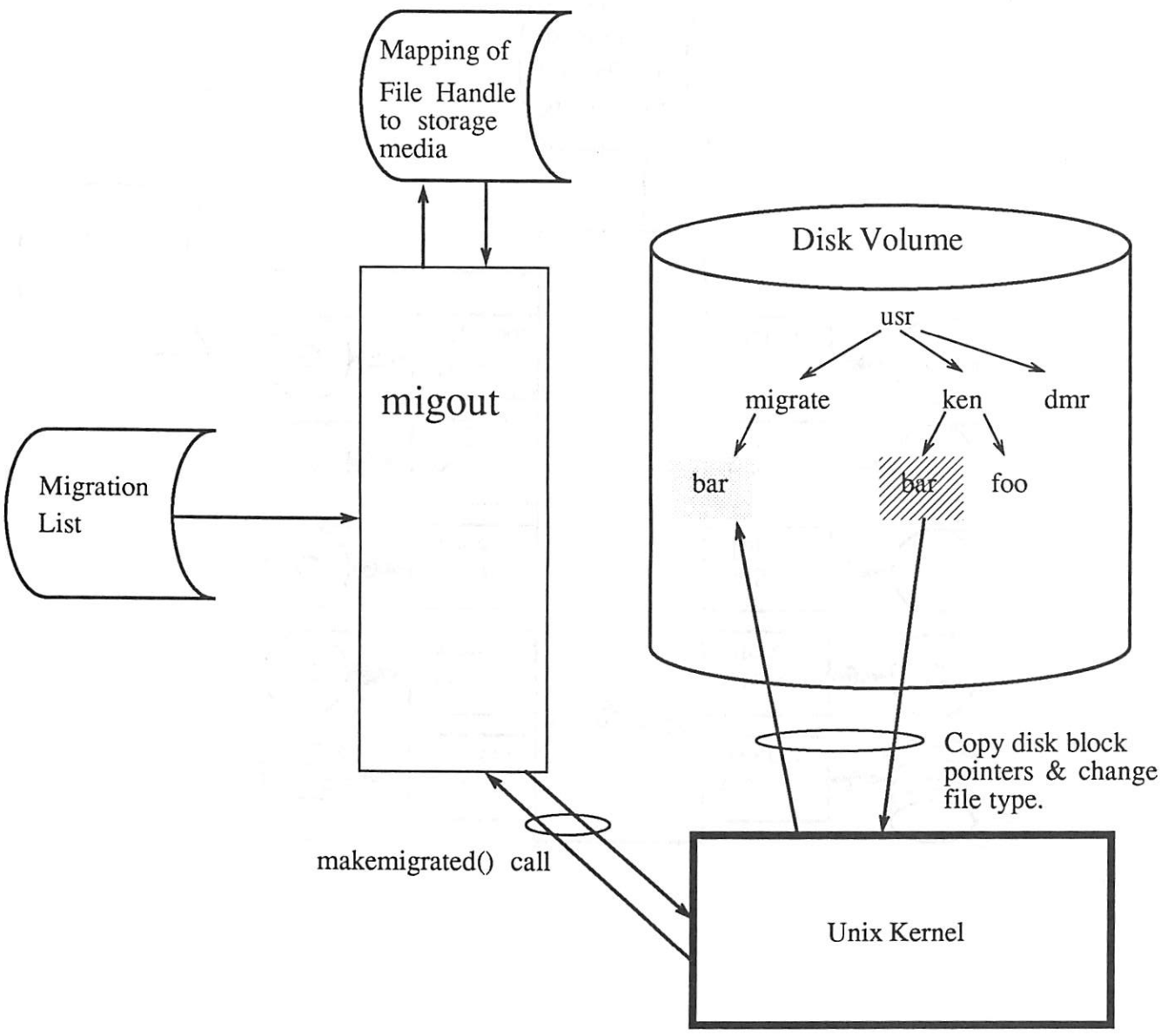
The authors would like to thank Chris Johnson for his careful and thorough review of the kernel code, Rick Matthews for his stimulating interchange of ideas, and Bob Reschly and Phil Dykstra for their advice, proofreading, and gratuitous humor as we designed and built this software.

The following strings that have been included in this paper are known to enjoy protection as trademarks; the trademark ownership is acknowledged in the table below.

Trademark	Trademark Owner
Cray	Cray Research, Inc
Ethernet	Xerox Corp.
FX/8	Alliant Computer Systems Corp.
IBM 370	International Business Machines Corp.
Macintosh	Apple Computer, Inc
NFS	Sun Microsystems, Inc
PowerNode	Gould, Inc
ProNet	Proteon, Inc
Sun Workstation	Sun Microsystems, Inc
UNIX	AT&T Bell Laboratories
UNICOS	Cray Research, Inc
VAX	Digital Equipment Corp.

Functional Diagram

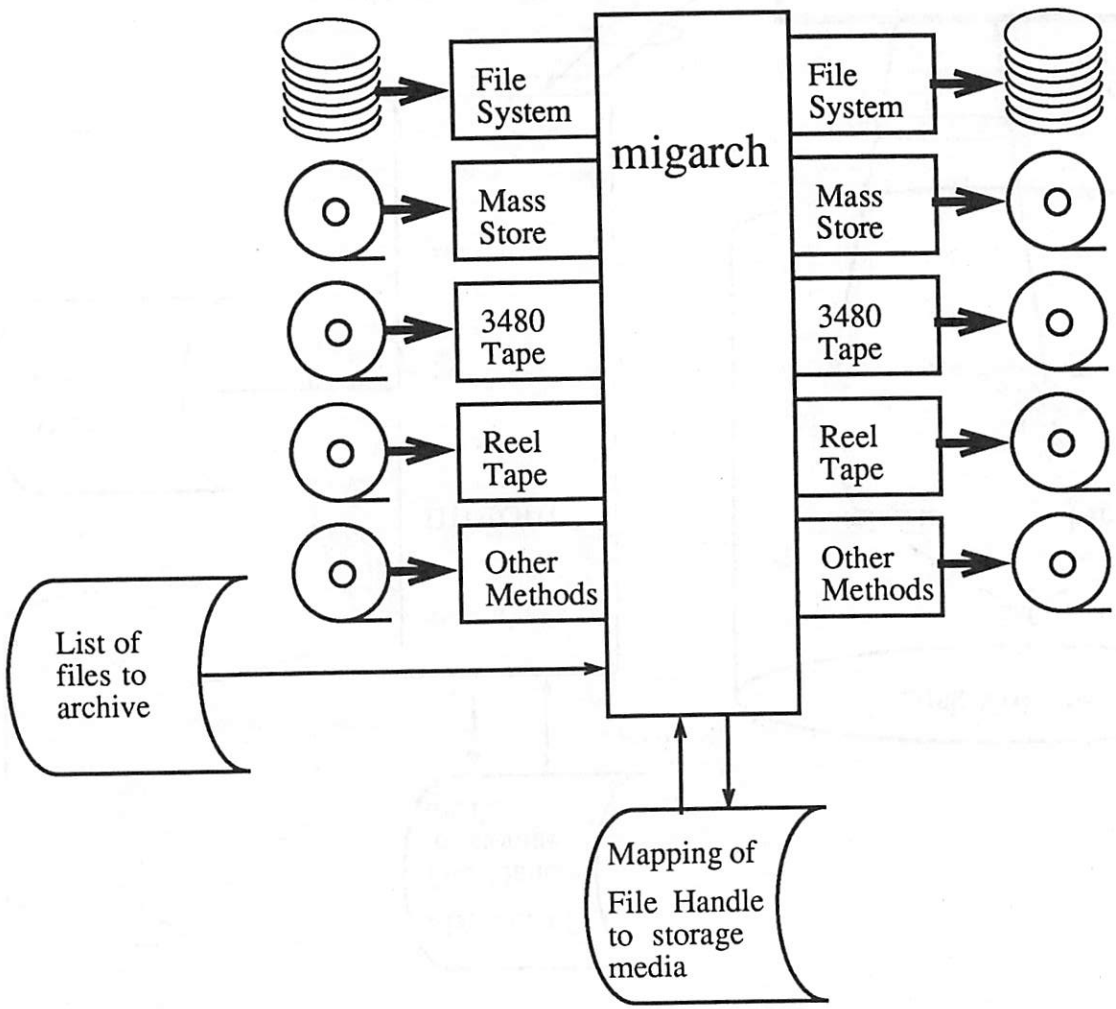
Out Migration (migout)



bar = Pre-migrated file bar = file's stub inode

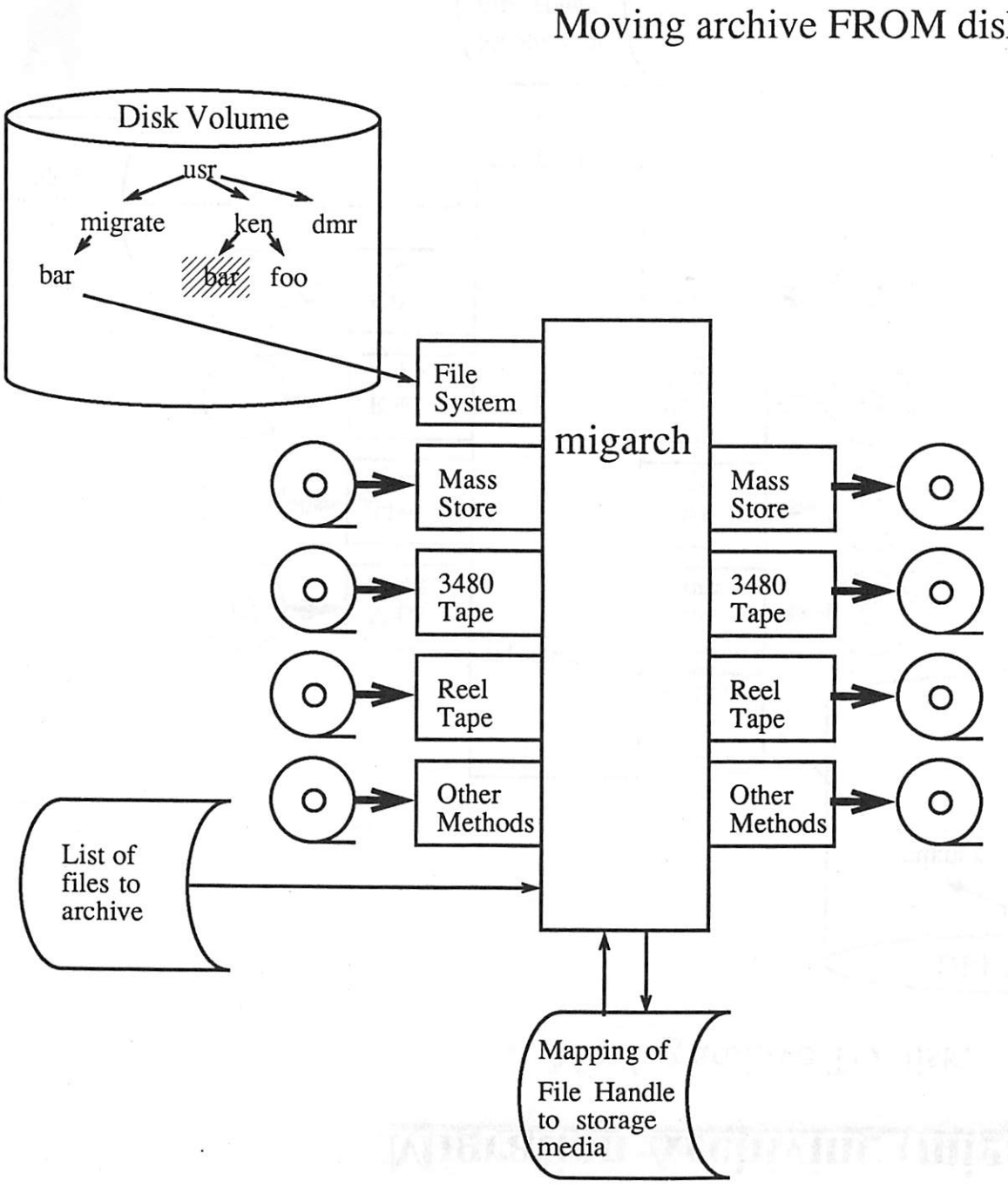
- Migout reads migration list
- For each entry, a kernel call is made to make the file 'migrated'. The kernel does this by creating a new file in the 'migrate' directory of the file system and copying the disk block pointers from the original file into this migrated file. The original file's inode type is changed to 'IFMIG' and its disk block pointers are cleared.
- The file handle database is updated with the current location of the file.
- A "tape header" file is also created in the 'migrate' directory to hold other associated information about the file. This header is to be written onto tapes and other backing storage media to ensure that some information about the file will always be directly associated with the image in the migration system.
- Because the file is still on the disk and can be quickly recovered, this is termed 'pre-migration'. However, the file is not truly pre-migrated until it has been copied to another archival method by migarch.
- In case of system crash, normal Unix backup utilities will be used to restore the file and/or the stub.

Migration Archiving (migarch)



- Migarch moves or copies archived files from one method/media to another method/media. The methods shown are typical of those at most sites. Other methods may include optical or mag tape auto-changers.
- External tools will build a list of files to archive, the destination method, and the number of copies to make using that method (i.e. copy from the staging area on disk [pre-migrated file] 3480 style magnetic tape and make two copies on the destination method.
- Different copies of a file are guaranteed to be on different media volumes.
- Methods may include data compression to achieve better storage utilization.
- Most methods will support recording in chunks of at most 10Mby each. Each chunk will have a volume, file, and chunk descriptor record written with it. This will allow easy recovery of data from damaged volumes.
- Must support media management and reclaiming partially used volumes. Update file handle DB with file's location info.

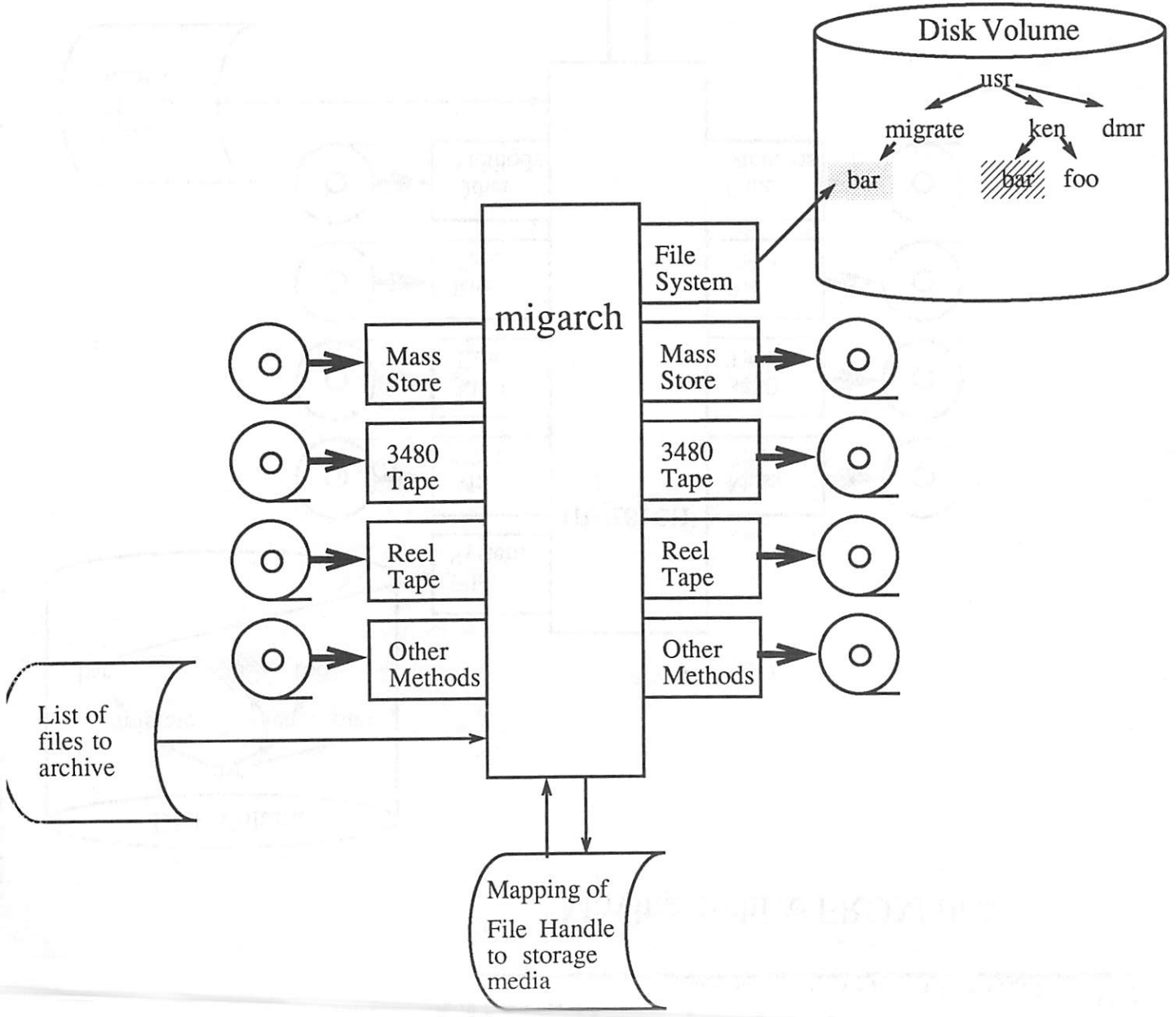
Migration Archiving (migarch)



- Methods may include data compression.
- Must support media management and reclaiming partially used volumes. Update file handle DB with file's location info.

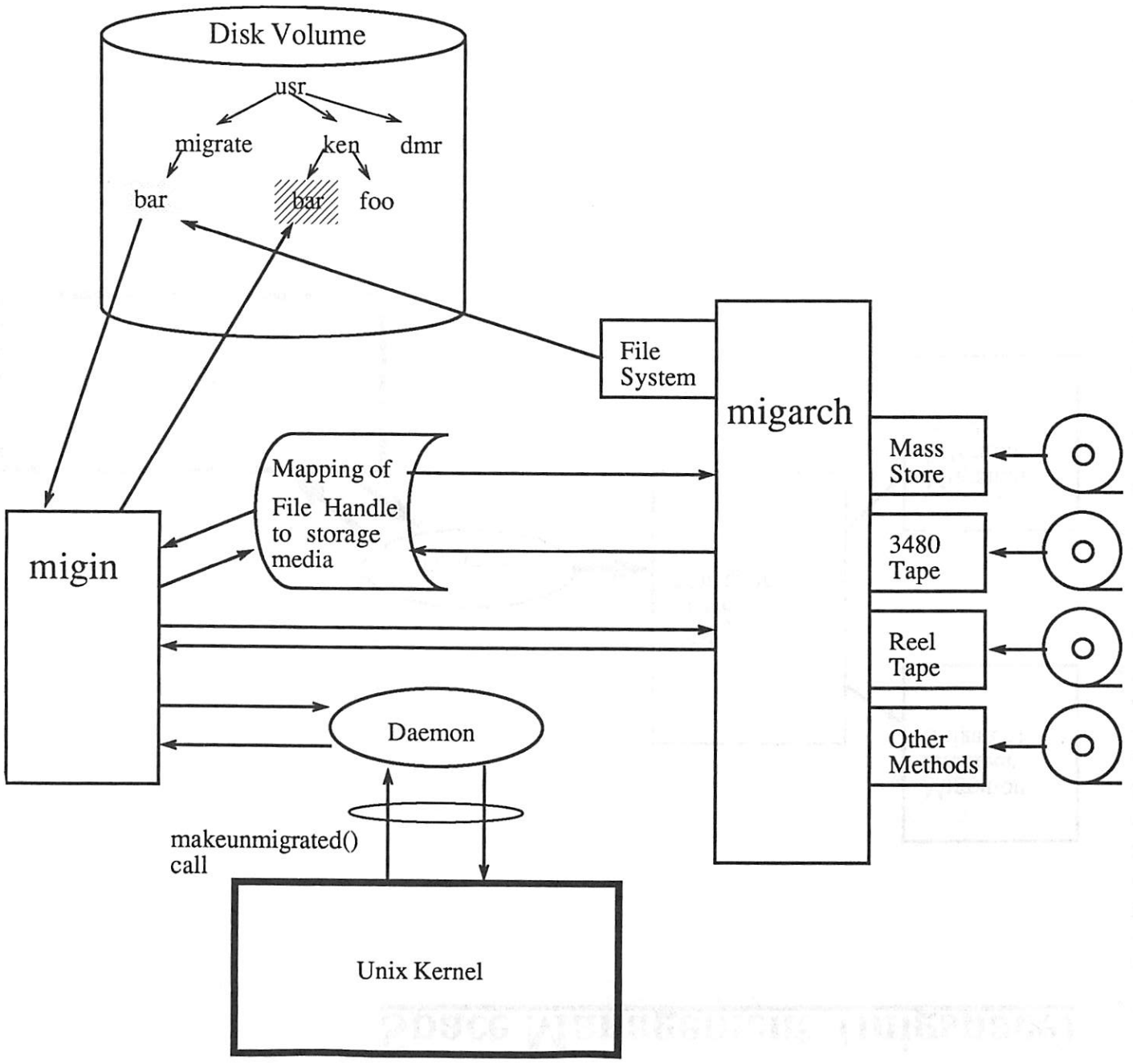
Migration Archiving (migarch)

Moving archive TO disk.



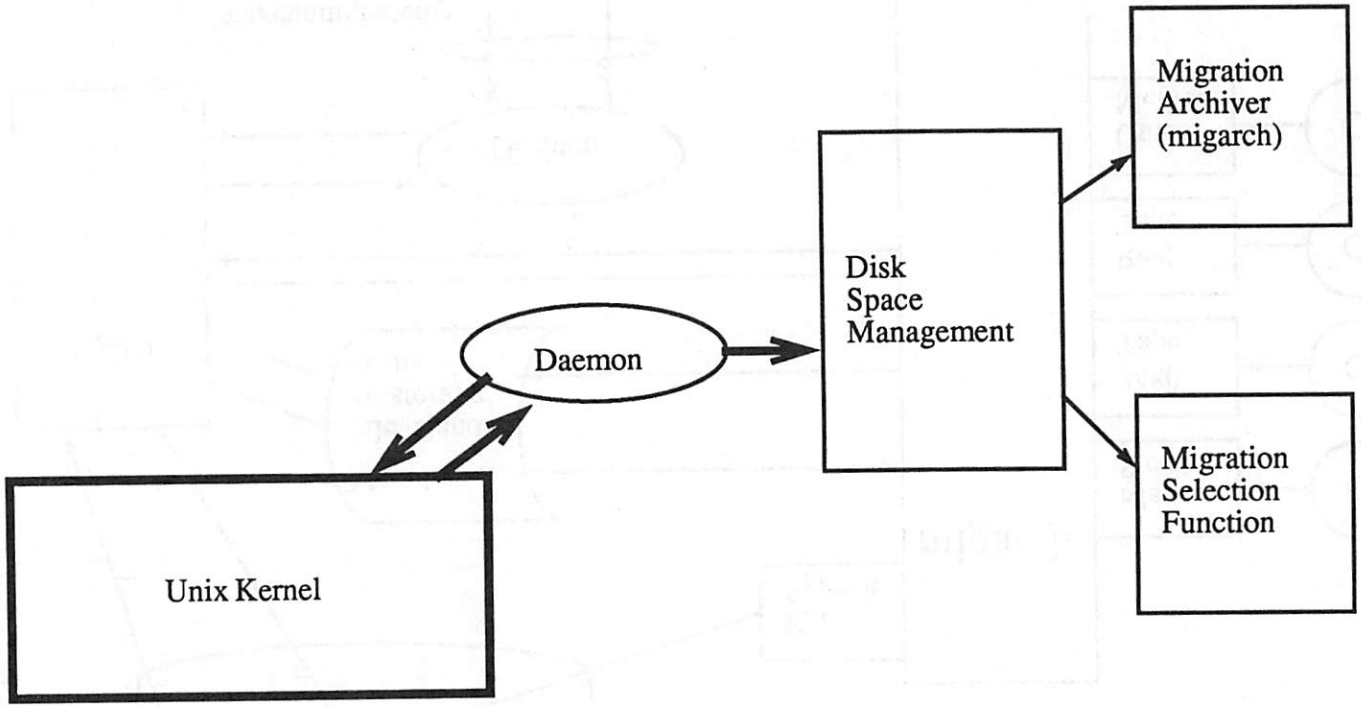
- Methods may include data compression.
- Must support media management and reclaiming partially used volumes. Update file handle DB with file's location info.

In-Migration Function (migin)



- Kernel requests a file reload from the Daemon. The Daemon starts migin to perform media selection and to control the archiver.
- Migin starts the archiver, migarch, to move the file from a backing storage media to the disk's staging area.
- Migarch handles media failures and automatically switches to alternate copies of the file stored on other media. Only if all copies fail will the user recieve a file open failure. Migarch sends an acknowledgement (success or failure) to migin when the reload is complete.
- Migin passes the confirmation to the Daemon which performs the inode swap (via the kernel) to cause the file's contents to be moved back into place.
- The FHDB is updated to indicate that the file has been reloaded. The entry in the FHDB will not be deleted at this point as a disaster recover aid. Another tool will perform database and media compaction by removing outdated entries.
- Migin keeps tapes mounted until some timeout interval expires or until the drive is needed for another request. Migin will die itself after being idle for a longer period of time.

Space Management (migspace)



- The kernel informs the daemon of pending disk space shortfall. The daemon starts space management facility (migspace). Migspace first checks for files which have been pre-migrated and have also been written to archival storage. Pre-migrated files matching this criteria are unlinked (note that the file's original stub inode remains in the file system, only the staged copy is unlinked.)
- The kernel informs the daemon of space exhaustion. The daemon starts the space management function which in turn starts the Migration Selection Function to begin searching the file system for migration candidates. Candidates are Pre-Migrated. It then starts the Migration Archiver to force these files out to backing storage. Only after the file has been written to another archive media is the pre-migrated file allowed to be deleted.
- The kernel is modified to allow processes to block until space is made available.
- The space management function should be very flexible to allow site dependent configuration. Different sites may want different selection criteria of which files to migrate.

A High Performance File System for UNIX

Alan Poston

GE Aerospace
NASA Ames Research Center

ABSTRACT

This paper describes a UNIX implementation of a High Performance File System (HPFS), which employs disk striping, volume seek ahead, and full tracking I/O for individual files. The new file system also allows multiple partial stripe I/Os to occur simultaneously, giving a significant fraction of the possible striping performance to randomly accessed files. The HPFS performs much better for sequentially accessed files, both small and large, and somewhat better for random accesses to large files and about the same as UNIX V5.3 for random access to small files. The goal of the HPFS design is to meet the demands of both the supercomputer industry, characterized by very large sequential files, and the transaction processing industry, characterized by large, randomly accessed files, while performing at least as well as current implementations on more traditional UNIX files, i.e. small sequential files. The design of the HPFS is described in detail, with emphasis on the ease of implementation. Reliability of the HPFS is characterized and future enhancements are outlined.

1. Introduction

The largest single barrier to the use of UNIX in an I/O intensive environment is its rather dismal performance in the area of file system I/O. This area has changed little over the years. The only common practice among UNIX implementers has been to increase performance by enlarging the file system block size [McKusi84] or by increasing the memory buffer cache [Smith85].

Hardware solutions to the file system performance problem have also been utilized, typically by introducing caching disk controllers[Grossm85].

Other solutions have included specialty file systems, such as the Contiguous File System[Ampex79]. These have provided some increase in file system performance but not without major drawbacks such as required preallocation and no file growth. The amount of performance gain in these systems is still limited by the performance of a single disk drive and controller combination.

File system striping is a relatively recent solution to the problem of performance. The concept is simple, spread the contents of a single file across multiple data paths, hence multiple disk drives, and achieve faster transfers by accessing the data in a parallel fashion. This type of file system is almost the opposite tack from a standard UNIX file system (as well as many other OS) where it is common to have multiple files systems on one disk drive. Striping file systems have been commonly seen in the supercomputer industry where files are characterized as being both large and sequentially accessed, traits which can fully utilize striping

performance. Cray Research Inc., ETA Systems Inc., and Thinking Machines Corp. all provide striping file systems of one kind or another.

The HPFS design marries the striping file system with data redundancy and the UNIX environment to provide a high performance standard UNIX file system.

1.1. Definition of Terms

A striping file system is one which performs simultaneous I/O on one file. This has traditionally taken the form of having multiple disk controllers, each controller supporting one or more disk drives. The controllers form independent data paths to and from memory. Portions of the file are spread across the disks on these separate data paths. The number of data paths (controllers) is also referred to as the stripe width. The number of disk drives on one data path is called the stripe depth. A stripe is the allocation unit which is spread across the stripe width. A striped block is a block from one disk on each data path. A striped track is a track from one disk on each data path. The striped file system can be viewed as a two dimensional array of disk volumes, where the columns are defined as disks which share the same data path and the rows are disks which have independent data paths (see Figure 1).

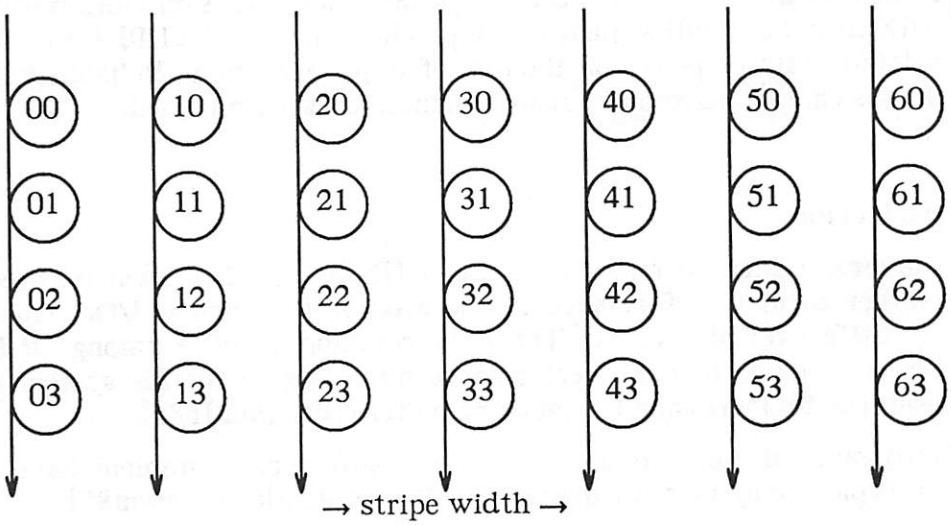


Figure 1: Striping File System Layout

Disk drives are units which can handle individual seek commands. Cylinders are made up of the individual actuator positions on a volume. Tracks are the various surfaces within one cylinder. Blocks are different rotational positions around each track. Note that for some disks, sectors are directly substitutable for blocks, this document uses blocks throughout.

A bitmap is a region of memory that describes the allocation of space on the file system. One bit contained in a bitmap describes some amount of space on the disk(s). A bitmap bucket is a piece of the bitmap used to allocate space for individual files, this is described in detail later.

It is assumed that the reader is already familiar with the concepts of superblocks, inodes and data blocks in a UNIX environment.

2. Design Overview

The overriding theme of this design is that this new file system should not be radically different than the current implementation. This helps the implementation because a great deal of code might be shared, which means that less code is added. This is also true for data structures such as buffers, superblocks and inodes. Others, [VanBaak87], have taken this approach to designing improvements to UNIX file systems. More specifically, the HPFS design calls for the use of standard sized blocks, shared use of the memory buffer pool, standard inode and superblock definitions, and even the sharing of the same device driver.

Four important concepts emerge from the design. First is the increase in performance by striping the I/O across multiple data paths, without loss of reliability. Second is the achievement of striping behavior by mapping logical block numbers onto physical blocks; this is critical to the design. Third is the allocation technique which allows for the maximum utilization of the available data path capacity. Fourth is the ability to detect sequential access operations and maximize performance by queuing plenty of read ahead blocks and, at a lower level, optimizing the current queue of requests into the minimum number of I/O operations.

2.1. Performance without loss of reliability

Performance is gained simply by having a number of disks, each with its own data path, participate in the transfer of data for 1 file. This is the basis of a striping file system.

A large problem with striping file systems is one of decreasing reliability with the addition of more disk drives. This problem has been explored by Patterson, et al.[Patter87]. They characterized the general solution as Redundant Arrays of Inexpensive Disks or RAID. They identified 5 members in the redundancy organiziation taxonomy, each member requiring different hardware and software configurations. They labeled these as level 1 through level 5 RAIDs.

The problem is stated by the following formula:

$$MTTF_{STRIPE} = \frac{MTTF_{DISK}}{\#DISKS}$$

where MTTF stands for Mean Time To Failure, usually a number provided by the manufacturer of the disk drive. As Patterson showed, the MTTF of a group of 100 disks whose individual MTTF was 30,000 hours (greater than 3 years) is only about 300 hours, or less than 2 weeks.

Briefly stated, the different levels of RAIDs combated the reliability problem by adding redundancy to the disk array. The level 1 RAID is simply a mirrored disk, which is expensive in both data path capacity and in disk space. The level 2 RAID employed a Hamming code error correction technique requiring less overhead resources, only 12% to 38% of the stripe size, but requiring that data be transferred a stripe at a time. This is similar to the memory organization of a computer employing SECDED. Level 3 RAID recognizes that detecting errors is (probably)

already built into the drive, so all that is needed is a check disk for the stripe, thus further reducing the overhead cost, but still having the restriction of stripe reads and writes. Level 2 and 3 RAID assume bit or byte interleaving across the stripe. This type of striping is explored by [Kim86] and performs better with synchronized disks. The fourth level RAID assumes block interleaving across the stripe. This allows for independent I/O to occur within the stripe and avoids the necessity of synchronized disks. The partial stripe I/O requires a read/modify/write cycle involving both the data to be replaced and the check disk for writes. The bottleneck is identified as the check disk, since it is involved in every partial stripe write. In the fifth and final level RAID, the check disk information is spread evenly across all members of the stripe. This allows for multiple partial stripe writes to occur simultaneously, although the read/modify/write operation is still required.

Level 1 RAID allows for the most I/O operations per second to occur as there is no check disk (every disk is mirrored). The drawback is the expense of halving the disk and data path capacity. Levels 2, 3, and 4 all perform fairly well (near data path capacity) for large sequential I/O operations, with levels 3 and 4 achieving more available disk capacity by only providing a single check disk string. However, they do not perform well at all on small random I/O operations, achieving only about 10% of available data path capacity. The level 5 RAID performs the same as 3 and 4 for large sequential I/Os, but is much better on small random I/O, achieving 60% of available data path capacity. These calculations were performed for a stripe width of 10 disks, other widths may perform differently.

The probability of failure of the group of redundant disks can be characterized as the probability of a second disk failure before the first failed disk has been repaired or replaced. With the single check disk on a stripe width of 10 and depth of 1 and a single disk Mean Time To Repair of 1 hour, the MTTF of the RAID is calculated to be 820,000 hours or better than 90 years, with a stripe width of 25 it is over 40 years, clearly exceeding any requirements.

The HPFS design employs a level 5 RAID approach, although the actual implementation is not on "inexpensive" disks. The acronym humorously bestowed on the HPFS implementation is RAVED for Redundant Array of Very Expensive Disks.

2.2. Logical Block Mapping of the HPFS

The mapping of logical blocks to physical blocks is the basis for achieving striping performance for single files while still allowing random block I/O to occur.

Components of the disk physical address form a 3-tuple whose elements are cylinder, track, and block. The traditional method of mapping logical blocks to the physical 3-tuple is to order the 3-tuple to disk[cylinder, track, block]. As an example, suppose a disk drive has 819 cylinders, 15 surfaces or tracks per cylinder, and 10 blocks per track, the mapping would have logical block 0 map to cylinder 0, track 0, and block 0 or disk[0,0,0]. Logical block 1 would map to cylinder 0, track 0, and block 1. Logical block 10 would map to cylinder 0, track 1, block 0; logical block 150 maps to cylinder 1, track 0, block 0; and so on. The reason to map a disk this way is to provide a minimum distance in both rotational delay and seek time to logical blocks which are numerically near each other. By understanding the

characteristics of the hardware involved, the ordering might change. This is most often seen as block interleaving, where the hardware is unable to finish a transfer and start another by the time the next rotational block comes under the head. A two to one interleaving scheme would have the mapping : LB 0 = disk[0,0,0]; LB 1 = disk[0,0,2]; LB 5 = disk[0,0,1]; and LB 9 = disk[0,0,9]; and so forth.

The HPFS design has 5 components of the physical disk address. These form a 5-tuple whose elements are data path or column, row, cylinder, track, and block. The last three have the same definition as the last example, the data path refers to different disks across the stripe width and the row refers to different disks down the stripe depth. There are now numerous different orderings to the mapping, many of which can provide a minimum time between numerically near logical blocks. Because blocks across the stripe have no delays from one another (simultaneous access), this then is the index which will be the last in the ordering, or vary the fastest. Blocks around the track are chosen for the second to last index. Rows are picked next. This is because the seek time on the next row can be hidden by the track's worth of blocks being transferred on the previous row. Picking the next track on the same disk achieves the same goal but has the detrimental effect of "locking up" a disk for N tracks of data, where N is the number of surfaces. If another I/O is allowed on the same disk before all the tracks are transferred then two seek time penalties are required. By placing the row next in the ordering, the amount of readahead need only be 2 or 3 stripe tracks worth of data, reducing the amount of system buffer space required and allowing other processes to access data without waiting for N rotations of the disk. The next component in the ordering is the tracks within the cylinder, and the first index in the ordering is therefore the cylinder within the disk. The ordered 5-tuple now looks like diskarray[cylinder, track, row, block, data path].

Examples of logical block mappings, using disks with 4 blocks per track, 4 data paths and 2 rows on each data path (8 disks in the array); LB 0 maps to data path 0, row 0, cylinder 0, track 0, block 0. LB 3 maps to data path 3, row 0, cylinder 0, track 0, block 0. LB 4 maps to data path 0 row 0, cylinder 0, track 0, block 1. LB 16 maps to data path 0, row 1, cylinder 0, track 0, block 0. LB 32 maps to data path 0, row 0, cylinder 0, track 1, block 0 (See Figure 2).

Remember from previous discussion that a level 5 RAID is the design model for the HPFS. This means that one of the blocks in each stripe is the check block. The check block needs to move from data path to data path in sequence in order to avoid bottlenecks on partial stripe updates. If a stripe allocation method is used then there is also a need to spread the starting block in the stripe across the data paths for the same reason. If all files started on data path 0, then small files, those less than a striped block in size, would tend to interfere with one another because they occupy the same data paths. By mapping the logical blocks so that if block N occurs on data path K then block N+stripe_width occurs on data path K+1 modulo stripe_width, both problems can easily be solved. The check block is now the last logical block in each striped block.

Now the logical block mappings are: LB 0 maps to data path 0, row 0, cylinder 0, track 0, block 0; LB 3 (a CHECK block) maps to data path 3, row 0, cylinder 0, track 0, block 0; LB 4 maps to *data path 1*, row 0, cylinder 0, track 0, block 1; LB 7 (a CHECK block) maps to *data path 0*, row 0, cylinder 0, track 0, block 1 (See Figure 3).

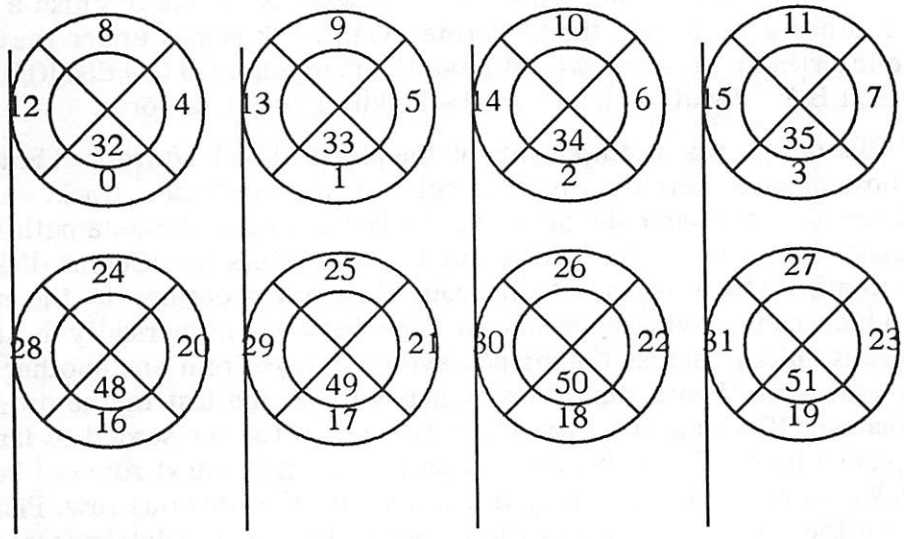


Figure 2: Logical Block Mapping

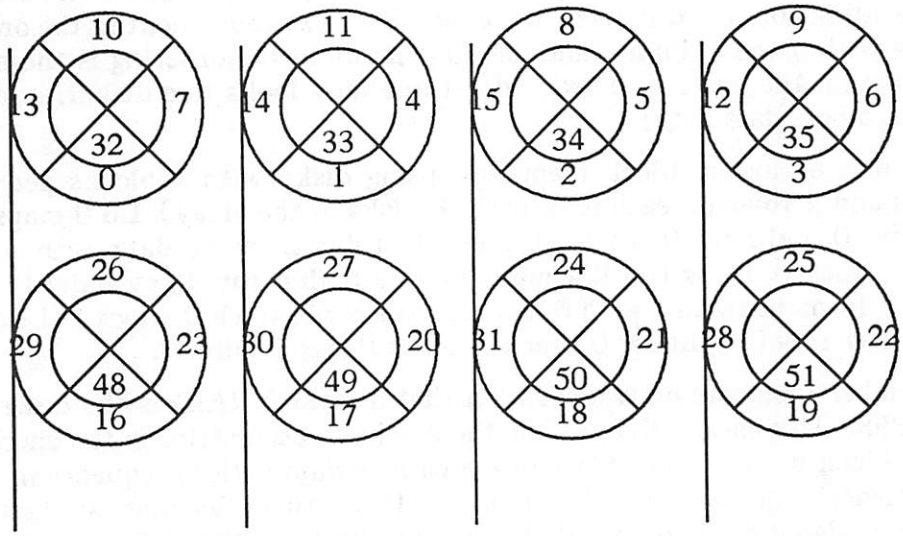


Figure 3: Logical Block Mapping with start block rotation

By choosing this mapping scheme, only 2 or 3 striped tracks of data need be requested at one time to achieve continuous striping performance. This statement is based on the assumption that file allocation occurs such that major portions of the file have sequential logical blocks and the assumption that the device driver layer can organize multiple I/O's to a disk in an optimal fashion.

2.3. New Allocation Scheme

As demonstrated in the last section, striping performance to a single file can only occur if major portions of the file are in sequential logical block order. The current V5.3 scheme of LIFO (Last block In to free list is the First block allocated or Out) allocation is inappropriate for this task. A much more appropriate approach is that taken by BSD 4.2 [McKusi84]. This approach breaks the disk into cylinder groups and allocates space for files from a cylinder group bitmap. The bitmap contains a bit for each block of data on the disk, it is marked as allocated by a 1 and free by a 0. If the file system needs N blocks sequentially ordered, it need only find a group of N 0 bits in the bitmap. It is much faster to scan the bitmap for a group of free blocks than it is to search a freelist.

The HPFS design does not need cylinder groups as defined in BSD 4.2, it does need a similar idea to cylinder groups, however, in order to continue allocation for a given file in the same striped track where it last allocated a block. In the HPFS design, the cylinder group is known as a bitmap bucket, where the bucket is the amount of the bitmap that represents a disk array track's worth of space. Another way of expressing this is to say a striped track by stripe depth amount of space or the last 3 components of the 5-tuple which make up the logical block mapping scheme. If the stripe width was 10 and the stripe depth was 5 and the number of blocks/track was 10 then the bitmap bucket would be 500 bits long (assuming 1 bit represents 1 block).

The allocation algorithm is simple. If a process begins to allocate space for a file and it currently is not assigned to a bitmap bucket then a new bucket is selected for this file. The selected bucket will be the one which has the most space and which is not currently in use. The bucket is marked as being in use until the file is closed. If the bucket runs out of space then a new bucket is assigned using the same selection criteria. The use of the bucket could be maintained across file closings by simply examining the logical block number of the last block allocated. This will directly relate to the last bucket used by this file.

A further refinement is possible. The bitmap bucket could be divided into sub-buckets, where each sub-bucket represents a striped track of blocks. Allocation by files which are new to the bucket should take place in empty sub-buckets. This allows for full tracking I/O to take place, because only 1 file occupies each striped track. If the file system becomes so full that no empty sub-buckets can be found, then non optimal allocation could occur, and files allocated after this occurs will not see the full tracking behavior, although they could still see striping behavior.

2.4. Striping Performance in a UNIX Environment

With the logical block mapping scheme and the bitmap allocation scheme, striping performance is shown to be possible in a UNIX file system. What is left is to describe how the HPFS design makes use of these features and actually achieves striping, full-tracking, volume seek ahead performance.

Striping performance is achieved by issuing the appropriate number of block I/O's (either by reading ahead or by writing behind) to cause a striped block of data to be transferred. Full track, striped performance is caused by issuing even more (a striped track) I/Os; and volume seek ahead, full track, striped

performance is caused by issuing even more I/O's. This then is a simple concept, if enough block I/O's are issued and if the allocation took place "correctly", the highest possible performance is achieved.

Before any of this occurs, UNIX needs to decide if it is appropriate to even attempt this. If the file is not being accessed in a sequential manner, all of this activity is wasted. In fact, the performance of the file system would suffer because of excess transfers and memory utilization. Fortunately, the decision about the type of access is easy, if the next block requested follows the last block requested, then the access is sequential and striping performance is attempted. If the next block requested is the first block in the file and there was no previous block then the access behavior is also deemed to be sequential. All other access patterns are deemed to be random behavior and no read ahead/write behind is attempted. This may not be the most accurate behavior evaluation and it could occasionally miss, but it is simple and will handle most cases.

After having decided that this is a sequential access and issuing the appropriate number of I/Os, how does striping occur? The Block I/O layer in UNIX queues the I/O to individual device queues, the device driver examines each devices' queue and, if the device is not busy, issues the I/O. Because of the logical block mapping and the allocation scheme already discussed, striping simply happens. Full tracking is another matter; the device driver will have to be modified to examine the device queue and make the determination that a full track I/O is possible. Full tracking is possible if all of the blocks which make up a single track on this disk appear on the I/O queue and, furthermore, they are all to be read or written (intermixing is not allowed). If these conditions are met then the driver can construct a single I/O to transfer the entire track, making use of known rotational position or, in some cases, simply ignoring the rotational position (soft formatting the track as it is being written). Volume seek ahead occurs as automatically as striping, i.e. if enough I/O is queued to the devices then the device driver will issue the I/O to both the current row and to the next row (next in the stripe depth). These disks can seek to right actuator position but they cannot start transferring until the previous row is finished using the data paths. As the current row finishes the next row can start transferring and a whole new set of read ahead / write behind blocks are queued to the yet another row of disks in the stripe depth.

By changing the file I/O layer in UNIX to recognize sequential access patterns, and by issuing the correct amount of read ahead / write behind block I/Os, and by changing the device driver to recognize full track I/O opportunities, the HPFS design achieves striping, full tracking, volume seek ahead performance.

2.5. Calculation and Placement of the Redundancy Information

As mentioned in section 2.2, each stripe has a check block associated with it. The check block rotates from disk to disk in order to prevent partial stripe updates from interfering with one another. The logical block mapping scheme takes care of this detail, providing a uniform look to the placement of the check block. The check block is always the largest logical block number in the stripe.

The check blocks cannot be allocated to any file, they must appear as "preallocated" space in the bitmap. As such, they are never scheduled for I/O operations in the normal sense.

The check blocks are not needed at all during normal read operations, full or partial stripe. However, during a full stripe write operation, the check block needs to be calculated and written. Even worse, during partial stripe updates, the check block *and the previous data block* need to be read and a new check block calculated and written. All of this activity occurs when UNIX has decided to reclaim a system buffer or flush dirty buffers.

The calculation of the contents of the check block is extremely simple. The blocks which make up the stripe are logically summed (exclusive or) with one another in much the same way a parity bit is calculated. If a disk has failed, the contents of the missing block can be reconstructed by performing the logical sum calculation on the remaining blocks, including the check block. If it is the check block which is missing, then when the repair is finished, the check block could be reconstructed. If the "keep it simple, stupid" approach is taken here, then the new disk would have the product of the logical sum operation of all the other disks in the row. This replaces both the missing data blocks and the missing check blocks contained on the failed drive.

2.6. Block I/O to Device Driver Communication

As mentioned previously, the block I/O layer needs to queue a large number of read ahead or write behind buffers to the device driver. This could be accomplished by making repeated calls to the routines which assign buffers and to the device driver strategy routine. This would follow the normal way that UNIX works. This approach has a few drawbacks; the device driver could start a transfer prematurely, before all of the blocks which make up a track are queued, thus eliminating the full track I/O opportunity. Another drawback is the shear amount of overhead involved in making repeated subroutine calls.

A better method is needed for the HPFS design. The HPFS design gathers a group of buffers from the freelist and queues them *en masse* to the device driver. This can be accomplished by making a few changes to the standard UNIX code. The buffers to be passed to the device driver are linked together using the av_forw and av_back pointers in the buffer header. These pointers are not used after the buffer has been removed from the freelist, although some implementations use these pointers in the device driver to form the queue of current I/O requests. In any case, they are not used between the time of freelist removal and the I/O queuing, so they can be used to form the chain of requests to the device driver layer. The actual call to the strategy routine remains the same. Other routines which pass buffers to the strategy routine will have to be changed to terminate the chain with the first buffer. The device driver will need to be changed to follow the chain of requests, queuing the buffers to the appropriate drives' I/O queue. After the last buffer is queued, then the non busy drives can have I/O started on them.

2.7. Buffer Management and the Check Block

There is a new concept which the buffer management layer needs to address. This is the concept of requiring a free buffer in order to actually write a dirty buffer to disk. The HPFS design could require this free buffer in two ways, either because a check block needs to be calculated or the previous contents of a block are needed for a partial stripe update.

UNIX keeps all buffers in the buffer pool on one of two disjoint lists. The two lists are the freelist and the awaiting I/O list. A buffer may only exist on one or the other of these lists. In addition, the buffer may be placed on a hash queue list. This list is used to speed the process of finding a cached block. Buffers may exist on the hash queue list and one of either the freelist or I/O list simultaneously. Various flags are marked in the buffer header to indicate the state of the buffer.

A process doing I/O will first attempt to find the required block in memory, if the block cannot be found then a buffer is requested from the freelist. If the freelist contains buffers marked for delayed write, the process schedules the buffer for I/O and continues searching. If the end of the freelist is reached, the process sleeps. The process will be awakened when a delayed write is finished and the buffer is returned to the freelist. HPFS has a problem with this scheme. If all of the buffers on the freelist are marked for delayed write, a deadlock situation occurs. The delayed write cannot start until a free buffer is available for either a check block or for a previous data block in a partial stripe update. If all of the buffers are on the freelist, then no new buffers will be placed on the freelist by I/O completion and no buffers will ever be available for the check block calculation.

The solution to the problem is to reserve a certain number of buffers in the pool which cannot contain delayed write blocks. This is implemented by counting the buffers in the pool which do not have the delayed write status. Every time a delayed write occurs, the count is decremented. If the count reaches a known fixed value, a stripe write of delayed buffers is started. As these I/O's start, the count is incremented. This is similar to what happens now in UNIX, except the count is implied to be zero by the code. The reserved value is determined as the minimum number of buffers needed to do at least one partial stripe update. For the worst case partial stripe write this value is the stripe width minus one plus one for the check block for this stripe. If the activity of the file system is such that delayed writes are sleeping waiting for the first few stripe writes to complete, then the number of reserved buffers should be increased. Note that these buffers are not reserved empty, they may still contain cached read data.

A second change is required for partial stripe updates. HPFS needs the old contents of a particular disk block in order to remove its' contribution to the existing check block, before adding the contribution of the replacement block. This implies that two buffers will exist in memory for the same disk block, the old and the new. A flag is needed in the buffer header to indicate that this situation has occurred to prevent accidental cache hits on the old block. Remarkably, a flag already exists in UNIX for a similar purpose, B_STALE. It implies a slightly different meaning however, indicating that an I/O error occurred for this buffer at some previous time. Using this flag for a new purpose may not be the proper way to address the issue.

3. Design Overview Wrapup

The design so far has shown that a striping file system in UNIX is possible and that, by using a level 5 RAID approach, it can achieve a significant level of striping performance for randomly accessed files without loss of reliability. The design has shown that the critical changes in UNIX to achieve this are 1) correctly mapping logical blocks to physical blocks, 2) a block allocation technique that causes a single file to occupy numerically close logical blocks, 3) recognizing sequential and random access patterns, and 4) organizing queued I/Os into a single full track I/O.

3.1. Planned Implementation of HPFS

This section of the HPFS design provides details of a planned implementation at NASA Ames Research Center. The Numerical Aerodynamic Simulation Facility at NASA Ames is one of the largest supercomputer installations in the world. The NAS project currently has two Cray 2 supercomputers, an ETA 10, a Thinking Machines Corp. Connection Machine, a collection of Convex class minisupercomputers, a large collection of Sun and SiliconGraphics workstations, a few Dec Vax's, and, finally, an Amdahl 5880. Almost all of the machines use some flavor of UNIX as the operating system, the Amdahl 5880 uses Amdahl supported UTS, with locally implemented BSD 4.3 modifications. The main task of the Amdahl is to provide Mass Storage support for all of the other machines, with particular emphasis on the HSP (High Speed Processors), currently the Cray 2s. To this end the Amdahl is configured to have 12 strings (data paths or channels) of 16 Amdahl 3880 double density disks. Each disk contains 1770 cylinders of 15 tracks, each track is soft formatted to 10 4K byte records. The total amount of storage in the disk farm is therefore 12 strings x 16 disks x 1770 cylinders x 15 tracks x 10 blocks x 4K bytes or about .21 TB (TeraByte). The goal of the file system is to provide a medium speed LAN with about 100 mbits per second of throughput capacity. A high speed LAN is planned for the future with capacities approaching 800 mbits per second. The major goal of the HPFS is to ensure that the Mass Storage System bottleneck is *always* the network. The MSS file system also has the ability to change media for a given file, placing the file on slower, cheaper, more dense storage when it fails to meet the criteria of high speed residency. This media change is transparent to the user, other than access to the file will be somewhat slower. The issues dealing with the media hierarchy are dealt with in other design documents and are not considered in the HPFS design.

References

[McKusi84].
M. K. McKusisk, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, vol. 2(3), pp. 181-197, August 1984.

[Smith85].
Alan Jay Smith, "Disk Cache - Miss Ratio Analysis and Design Consideration," *ACM Transactions on Computer Systems*, vol. 3, no. 3, pp. 161-203, August 1985.

[Grossm85].
C. P. Grossman, "Cache - DASD storage design for Improving System Performance," *IBM Systems Journal*, vol. 24, pp. 316-334, 1985.

[Ampex79].
Ampex Corporation, *Contiguous File System*, 1979. Internal Design Document

[VanBaak87].
Thomas VanBaak, "Virtual Disks: A New Approach to Disk Configuration," *Winter 1987 USENIX Association Conference Proceedings*, Washington, D.C., 1987.

[Patter87].
D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks," *Report No. UCB/CSD 87/391 Dept. of Electrical Engineering and Computer Science*, University of California, Berkeley, 1987.

[Kim86].
Michelle Y. Kim, "Sychronized Disk Interleaving," *IEEE Transactions on Computers*, vol. C-35, no. 11, November 1986.

Attaching IBM Disks Directly to a Cray X-MP

Douglas E. Engert

Argonne National Laboratory
Argonne, Illinois

ABSTRACT

This paper describes a method of attaching IBM 3380 disks directly to a Cray X-MP via the XIOP with a BMC. The IBM 3380 disks appear to the UNICOS operating system as DD-29 disks with UNICOS file systems. IBM 3380 disks provide cheap, reliable large capacity disk storage. Combined with a small number of high speed Cray disks, the IBM disks provide for the bulk of the storage for small files and infrequently used files. Cray Research designed the BMC and its supporting software in the XIOP to allow IBM tapes and other devices to be attached to the X-MP. No hardware changes were necessary, and we added less than 1,000 lines of code to the XIOP to accomplish this project.

Having acquired a Cray X-MP/14 compter in November 1987, Argonne National Laboratory faced the probability of a significant disk shortage in the coming year. A number of solutions were possible to alleviate the expected shortage: (1) buying more Cray disks, (2) relying more heavily on front-end systems that would act as file servers, and (3) finding a method of attaching other manufacturers' disks to the X-MP. All three of these methods were viable, and we began parallel efforts on each.

Attaching IBM 3380 disks to the X-MP was the most attractive solution to the problem. These disks are reliable, provide large amounts of storage, are cheap, and offer good performance. The configuration of the X-MP and the currently installed IBM 3380 disks at Argonne allowed us to start the project with no additional hardware costs. All the hardware needed was available, currently installed, and operational.

The primary design goal of this project was to keep it simple. We wished to change as little code as possible in UNICOS. We would use the disks to hold UNICOS file systems, and we would not share the disks with other systems. To the user, these file systems would appear as any other UNICOS file system. Small files and infrequently used files would be stored on these disks. This procedure would complement the use of the existing Cray disks for the larger high performance user and system files. Thus the Cray disks and IBM disks would complement each other to provide high performance and large capacity.

The Argonne configuration consisted of a Cray X-MP/14 running UNICOS 3.0.11 with an I/O Subsystem (IOS) model-C. Included in the IOS is an XIOP, which is the processor in the IOS used to handle tapes. IBM and IBM-compatible tapes are attached to the XIOP by the Block Multiplexer Controller (BMC).

The BMC consists of four channel interfaces that attach to IBM compatible controllers via standard bus and tag cables. To the controllers, they appear as IBM

channels. The BMC-5 that we have is capable of running in data streaming mode and can transfer data at 3 Megabytes/second. These capabilities are normally necessary for IBM 3380 disks (operation with a BMC-4 may also be possible; see below). The Cray documentation indicates that the BMC was designed for the operation of IBM-compatible control units and not just for tape control units. All IBM channel commands are possible, and the channel can run in byte or block multiplexer mode.

The software in the XIOP was also designed to work with more then just tapes. There are two major subsystems in the XIOP. The TAPE EXEC subsystem handles all tape related processing, and it uses the BMX subsystem to do the I/O. Communication between the two systems is done with control blocks. This interface looks very similar to the EXCP interface of the IBM MVS operation system, where channel commands are passed to the I/O Supervisor. In IBM system terminology, the BMX code is a combination of the I/O Supervisor and channel. It has the concepts of device queuing, multipath access to devices, PCI interrupts, channel retry, command chaining, and interrupt processing.

Unlike an IBM system, the BMX has to do in software what the IBM system would do in hardware. The BMC implements the minimum amount of the channel protocols that need to be in hardware, and the rest of the protocols are implemented in software via the BMX subsystem. For tapes, this procedure is not difficult, since their channel commands are simple, but for disks, this extra software overhead can cause performance problems.

The flexibility of the BMC and the BMX software made the project of attaching the IBM 3380 disks to the X-MP possible. By using the BMX interface to do the I/O and its associated queuing, we could write an interface that consisted of about 1000 lines of code, most of which was contained in four new modules.

To the UNICOS system, these disks appear as DD-29 disks located on the XIOP. Since a DD-29 is slightly bigger than a single density 3380, only the first 600 cylinders of the DD-29 are defined to UNICOS. I/O requests to the IOS are sent via packets. These packets contain the cylinder, head, and sector numbers, the data address in Cray main memory, the number of sectors, and the read or write code. When one of these packets arrives in the XIOP, our code intercepts it and maps the DD-29 cylinder, head, and sector numbers to the 3800 cylinder, head, and record numbers. Our code then builds a channel program and invokes the BMX code to read or write the data. When the I/O operation is complete, the packet is sent back to the Cray as if the I/O operation had been done on a real DD-29.

Defining the IBM 3380 disk drive to the IOS required that additional units be defined in *confbmx.c*. This is the file that contains the definitions of the tape devices. The disks appear as a similar device with a different device type.

Limitations caused by the processing of channel protocol in software (in the BMX code) rather then in hardware have produced some performance problems. The normal method of doing I/O on an IBM disk is to use a set of channel command words that use a search command to find a record, followed by a read or write data command. The BMX software is not capable of getting the read or write command out to the device fast enough after the search completes, and a full

revolution of the disk is necessary to get the data. By using another method consisting of the define extent and locate commands, the BMX code is capable of searching and reading on the same revolution. The define extent and locate commands are available on the IBM 3880-3 controller with the speed matching buffer feature and with cache controllers such as the IBM 3880-23 (we have not tested the 3880-23). The speed-matching buffer feature will operate at 1.5 MB/s or 3 MB/s. We have tried it at both. Argonne has the BMC-5, which can operate at either speed. Installations having a BMC-4, which operates at 1.5 MB/s could use the IBM 3880-3 with the speed-matching buffer.

Since the disks we are using are attached to both the X-MP and an IBM MVS system, we have formatted the disks from the IBM system. The format program writes 4096 byte records, 10 per track. Each record corresponds to one DD-29 sector. The first cylinder of the disk has an IBM label and other control blocks, so that, in a shared environment, the IBM system can recognize the disk as having one large MVS file.

We have added interleaving and read ahead as performance improvements. A number of factors necessitate these improvements. Since the XIOP does not have a direct connection to Cray central memory, additional time is necessary to move data by using the other IOPs in the IOS. UNICOS does not always queue multiple packets for sequential I/O and will issue the subsequent request only when the previous request is complete. We have experimented with different interleave factors and have found that a factor of three would give the best performance. This factor gives a sustained maximum transfer rate of 0.78 MB/s. If a cache controller were used, interleaving would not be necessary, an a transfer rate on the order of 2.3 MB/s could be obtained.

We are currently conducting stress tests by placing a number of select users onto a few disks with two channel paths attached. We are continuing to make improvements in performance and error recovery. The project began in March when two of us took the Cray IOS internals class. Since then we have put in about three work months of effort. Future effort on this project will depend on the interest shown by others in this project and on our need for more disk storage. Certainly, this project has demonstrated the flexibility of the Cray X-MP design and the viability of alternatives for disk storage on Cray Supercomputers.

NOTES

USENIX Association Services and Benefits

The USENIX Association is a not-for-profit organization of individuals and institutions with an interest in UNIX and UNIX-like systems and the C programming language. It is dedicated to fostering the development and communication of research and technological information and ideas pertaining to and UNIX-related systems. The Association sponsors workshops and semiannual technical meetings, produces and distributes a bimonthly newsletter, *;login:*; publishes a quarterly technical journal, *Computing Systems*; and serves as coordinator of a software exchange via its "Software Distribution Tapes."

The Association was formed in 1975 and incorporated in 1980 to meet the needs of UNIX users and system maintainers who met periodically to discuss problems and exchange ideas concerning UNIX. It is governed by a Board of Directors elected biennially.

The USENIX Association offers several services to its members:

- Mailing of the newsletter *;login:*;
- Mailing of technical journal *Computing Systems*;
- Offering of various UNIX publications and technical information for purchase;
- Presentation of technical meetings twice a year and single-topic workshops periodically;
- The right to order 4.3BSD UNIX Manuals;
- A discount on the meeting registration fee;
- The right to vote on matters affecting the Association, its bylaws, and in the election of its directors and officers.

STUDENT MEMBERSHIP $15

Open to any full-time student at an accredited educational institution. A copy of your student I.D. card must be provided.

INDIVIDUAL MEMBERSHIP $40

Open to any individual or institution. Individual Members may vote; however, they do not automatically receive the Distribution Tapes or other services requiring UNIX license verification.

USENIX Association Membership Application

Membership is by Calendar Year

Please type or print

[ ] New [ ] Renewal

Name: _____

Address: _____

_____

_____

_____

Phone: _____

uucp network address: *uunet!* _____

Individual, Corporate, and Supporting categories are all open to either institutions or individuals. Membership fees are:

[ ] $ 40 Individual

[ ] $ 15 Student (full-time)
With copy of student I.D. card

[ ] $ 275 Corporate

[ ] $125 Educational Institution

[ ] $1000 Supporting

[ ] Check enclosed: $ _____
Payments must be in US dollars payable on a US bank

[ ] Purchase order enclosed; invoice required

[ ] Check if you do **NOT** want your name and address made available to other members.

[ ] Check if you do **NOT** want your name and address made available for commercial mailings.

Please complete and return this form with your purchase order or payment to:

USENIX Association
P. O. Box 2299
Berkeley, CA 94710

For Office Use

Inst: .....

Mem#: ..... Check#: .....

Lic: ..... Rf: .....

Date: ..... Db: .....

USENIX Association
P.O. Box 2299
Berkeley, CA 94710